NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ANALYSIS OF TRAFFIC SIGNALS ON A SOFTWARE-DEFINED NETWORK FOR DETECTION AND CLASSIFICATION OF A MAN-IN-THE-MIDDLE ATTACK

by

Julian N. D'Orsaneo

September 2017

Thesis Co-Advisors:	Murali Tummala
	John C. McEachen
Second Reader:	Bryan Martin

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2017	**3. REPORT TYPE AND DATES COVERED** Master's thesis	
4. TITLE AND SUBTITLE ANALYSIS OF TRAFFIC SIGNALS ON A SOFTWARE-DEFINED NETWORK FOR DETECTION AND CLASSIFICATION OF A MAN-IN-THE-MIDDLE ATTACK			**5. FUNDING NUMBERS**
6. AUTHOR(S) Julian N. D'Orsaneo			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB number ____N/A____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Software-defined networking (SDN) has the potential to revolutionize the management capabilities of a highly distributed military communications environment. Yet, military adoption of SDN is contingent on a thorough analysis of security implications. In this thesis, we investigate a man-in-the-middle (MITM) attack that exploits the centralized topological view critical to SDN operations. In particular, we present a new scheme for detection and classification of the attack at the network layer. We apply wavelet analysis to detect anomalous conditions introduced by the MITM attack at traffic signals collected at network switch ports. Furthermore, we identify unique characteristics of reported anomalies in the collected traffic signals to build a classification framework. Other cyber events, such as a distributed denial-of-service attack and network congestion, are presented to the detection scheme to validate its general applicability. Overall, we successfully demonstrate the capability to detect and classify the MITM attack in addition to other cyber events at the network layer, thereby contributing to the security of SDN.

14. SUBJECT TERMS software-defined networking, network monitoring, wavelet analysis, anomaly detection, man-in-the-middle attack, anomaly classification			**15. NUMBER OF PAGES** 105
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

ANALYSIS OF TRAFFIC SIGNALS ON A SOFTWARE-DEFINED NETWORK FOR DETECTION AND CLASSIFICATION OF A MAN-IN-THE-MIDDLE ATTACK

Julian N. D'Orsaneo
Captain, United States Marine Corps
B.S., Villanova University, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2017

Approved by: Murali Tummala
Co-Advisor

John C. McEachen
Co-Advisor

Bryan Martin
Second Reader

R. Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Software-defined networking (SDN) has the potential to revolutionize the management capabilities of a highly distributed military communications environment. Yet, military adoption of SDN is contingent on a thorough analysis of security implications. In this thesis, we investigate a man-in-the-middle (MITM) attack that exploits the centralized topological view critical to SDN operations. In particular, we present a new scheme for detection and classification of the attack at the network layer. We apply wavelet analysis to detect anomalous conditions introduced by the MITM attack at traffic signals collected at network switch ports. Furthermore, we identify unique characteristics of reported anomalies in the collected traffic signals to build a classification framework. Other cyber events, such as a distributed denial-of-service attack and network congestion, are presented to the detection scheme to validate its general applicability. Overall, we successfully demonstrate the capability to detect and classify the MITM attack in addition to other cyber events at the network layer, thereby contributing to the security of SDN.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	application programming interface
DDoS	distributed denial-of-service
DoS	denial-of-service
DSTB	deployable site transport boundary
DWT	discrete wavelet transform
HP	Hewlett Packard
LLDP	link layer discovery protocol
MAC	medium access control
MITM	man-in-the-middle
OXM	OpenFlow extensible match
RTT	round-trip time
SDN	software-defined networking
SPP	synthetic-packet pairs
UDP	user datagram packet

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Dr. Murali Tummala, Dr. John McEachen, CDR Chad Bollmann, and LT Bryan Martin for their advice and guidance through this process. I would also like to thank my wife, Rachel, and son, Dominic, for their constant support and sacrifices.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Software-defined networking (SDN) brings increased adaptability, visibility, and local security to network operations through the separation of the control and data planes [1], [2]. The benefits offered by SDN align with the requirements of geographically distributed tactical networks employed in military environments. The employment of SDN within the framework of the military communications environment has the potential to alleviate the burden placed on tactically focused ground combat forces for network management [1]. With SDN, a geographically distributed network is able to be managed from a centralized location, away from the dangers of the battlefield.

The United States Marine Corps has sought to remove the complexities surrounding the establishment of tactical network environments through the introduction of the deployable site transport boundary (DSTB) [3]. The DSTB system is a preconfigured, deployable suite of servers and routers that enable connection to the Marine Corps' primary network. In reality, it does not represent a plug and play solution as intended. In contrast, SDN provides a true plug and play solution for the deployed network environment due to its capabilities to change network configurations from a central location with software [1], [2]. While centralized network management through the adoption of SDN produces desirable military benefits, it also introduces critical vulnerabilities [1], [2], [4].

A. BACKGROUND AND MOTIVATION

In addition to common network vulnerabilities, vulnerabilities associated with centralized network management have been consistently emphasized as a primary area of concern in SDN deployments [1], [2], [4]. Hong et al[4]. exploit the critical vulnerabilities introduced by centralized management in SDN to launch a man-in-the-middle (MITM) attack. SDN manages the network based on a centralized view of the network topology [4]. The success of their MITM attack relies on the manipulation of the centralized topology view by relaying Link Layer Discovery Protocol (LLDP) packets between attacking hosts [4].

The motivation of this thesis is to contribute to the hardening of the SDN environment through detection and classification of cyber events, such as the MITM attack designed by Hong et al[4].. SDN vulnerabilities introduced and exploited by Hong et al. compromise the integrity of the SDN topology view [4]. Such security risks need to be mitigated in order to pave the way for military adoption of SDN.

B. OBJECTIVE

The solution presented by Hong et al. to detect their proposed MITM attack focuses on fingerprinting network hosts at the SDN's central management location based on traffic at the application layer [4]. LLDP packets received from ports associated with fingerprinted network hosts reveal the attackers. Hong et al. acknowledge the need to analyze physical layer characteristics in order to detect the attack in the event the attackers try to hide. The attackers will be able to hide from their detection scheme if they prevent the generation of application layer traffic at the compromised attack hosts [4]. Hong et al. ignored this scenario in their work. In this thesis, we consider the attackers' capabilities to hide and attempt to make it more challenging for them to do so. While it is tedious to analyze traffic signals at the physical layer due to the requirement of physical network taps [4], [5], it is fairly straightforward to obtain and analyze traffic signals at the network layer using software based network management solutions.

The primary objective of this thesis is to present an alternative detection scheme at the network layer for the SDN MITM attack proposed by [4] in order to limit the abilities of the attackers to mask their attack. The secondary objectives are to classify the MITM attack based on the reported outputs of the detection scheme and generalize the capabilities of the detection scheme to detect other well-known cyber events. Our proposed solution relies on previous research presented by [6]–[8] in which wavelet analysis of traffic signals is used to expose anomalous network conditions. By collecting traffic signals at the network layer, we intend to extend the application of wavelet analysis to expose the anomalous conditions introduced by the MITM attack and determine if the identified anomalies are discernable from those introduced by other cyber events.

In addition to the SDN MITM attack, the cyber events used for validation of the detection scheme include a distributed denial-of-service (DDoS) attack and network congestion. The cyber events are emulated in a physical SDN environment and collected traffic signals associated with each event are analyzed by the detection scheme. The detection scheme is implemented in MATLAB.

C. RELATED WORK

The security solution implemented in [4] for the detection and prevention of the SDN MITM attack focused on monitoring application layer traffic to fingerprint hosts on the network. The relaying of LLDP packets by known fingerprinted hosts represented an anomalous condition, thus exposing the MITM attack. Their attack solution is included in a security application titled TopoGuard. TopoGuard also provides security solutions for other SDN network topology based attacks [4]. In this thesis, we recreate the MITM attack proposed by Hong et al. in a physical SDN environment and provide a solution for detection at the network layer rather than the application layer.

Wang et al[9]. investigated the MITM attack presented by [4] under the same conditions proposed in this thesis. They were concerned with detection of the attack when the attackers attempted to hide by not generating application layer traffic. In that scenario, TopoGuard fails to detect the attack [4]. Their proposed solution relies on the detection of increasing delay of received LLDP packets at the controller [9]. If the round-trip times (RTT) of LLDP packets observed by the controller exceed a previously established norm, it is assumed that they are being relayed inappropriately throughout the network, and the MITM attack is exposed. In contrast to their approach, we collect and analyze byte and packet data at all network switch ports and delay data at all communicating hosts for detection of anomalous conditions. Our detection scheme is capable of detecting and classifying the MITM attack in addition to other well-known cyber events. The approach taken by Wang et al. is solely concerned with detection of the MITM attack.

Dhawan et al. [10] proposed and validated an SDN security application titled SPHINX designed to expose SDN network topology attacks such as the MITM attack.

The SPHINX security application detects anomalous network conditions based on comparisons of pre-established policies with a well maintained flow graph of observed network traffic. Similar to our approach, SPHINX also accounts for anomalous conditions arising from reported switch port traffic statistics [10]. The switch port measurements account for a single component of their detection scheme, whereas in this thesis traffic data based on byte, packet, and delay serve as the primary inputs to our detection scheme. Furthermore, we use wavelet analysis instead of policies and flow graphs to reveal anomalous conditions in traffic signals generated with byte, packet, and delay data.

Wavelet analysis of traffic signals for the detection of anomalies has been widely researched and validated. Both online and offline anomaly detection methods utilizing wavelet analysis were presented in [6]–[8]. Our research primarily relies on the online detection method proposed by [8] in which wavelet analysis was conducted over network traffic signals using sliding windows, and anomalous conditions were exposed by comparisons with statistically established threshold values. The threshold values were determined from a wavelet analysis of normal traffic conditions. Some of the anomalous conditions considered for detection in [6]–[8] were introduced in conjunction with denial-of-service (DoS) attacks, flash crowds, and network congestion. None of the network events or attacks considered for anomaly detection included a MITM attack. Additionally, the detection methods did not attempt to further classify the events or attacks. The traffic signals processed for anomaly detection in [8] were generated with packet header data. The traffic signals processed for anomaly detection in [6] consisted of byte, packet, and flow data. In our research, we extend the application of wavelet analysis to the detection of a MITM attack and introduce attack classification into our scheme based on reported anomalous conditions. Furthermore, we introduce delay data as another traffic signal input to our detection scheme.

Lakhina et al. [11] detected anomalies for numerous network events and attacks such as DoS attacks, DDoS attacks, and flash crowds and proposed criteria for classification of the detected events and attacks. They analyzed byte, packet, and flow data for detection and localization of anomalous conditions related to the investigated

events and attacks. They classified the anomalies based on the traffic representations in which they were reported and their associated timing information. The classifications were validated through an observation of the reported anomalous characteristics [11]. They did not, however, investigate or present criteria for the classification of a MITM attack. In this thesis, we use the overall method introduced by [11] to present classification criteria for a MITM attack, DDoS attack, and network congestion. Our proposed classification criteria are solely based on the traffic representations in which the anomalous conditions were reported and do not rely on any observed characteristics of the anomalies for validation.

D. ORGANIZATION

This thesis is organized as follows. In Chapter II, we present background information related to the implementation of SDN, the cyber events investigated in this research, and the wavelet analysis method for anomaly detection. In Chapter III, we introduce the proposed detection and classification scheme. In Chapter IV, we detail the methods for emulating the cyber events in our physical SDN environment and present the results obtained from the detection scheme. In Chapter V, we provide conclusions based on a summary of the results and discuss areas for future work. In the appendices, we present the scripts used for realization of the MITM attack and implementation of our proposed detection scheme.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

The introduction of increased software solutions and centralized management to the networking environment through the adoption of SDN has presented numerous security risks. The MITM attack proposed by Hong et al [4]. and explored in this thesis takes advantage of security risks resulting from the centralized nature of SDN. In this thesis, we seek to provide a detection scheme for the MITM attack at the network layer by leveraging the capabilities of SDN and wavelet analysis. In this chapter, we present the key components of SDN. We introduce the strategic details pertaining to the realization of the SDN specific MITM attack and the additional cyber events tested within our detection scheme. Lastly, we provide an overview of wavelets and their applicability to anomaly detection. Wavelets represent the primary analysis tool used for detection of the MITM attack in our proposed solution.

A. SOFTWARE-DEFINED NETWORKING OVERVIEW

SDN removes the responsibility for routing decisions from individual, distributed networking components through a separation of the control and data planes [1]. Routing decisions, termed flow rules, are passed down to corresponding data plane switches by a central controller [1], [2]. The flow rules are maintained at each switch in flow tables and accessed for appropriate forwarding actions upon receipt of network traffic [2]. Desired actions are applied to received network traffic based on match and priority criteria defined in the flow rules [2]. Since flow rule actions determine packet forwarding, medium access control (MAC) addresses are not updated as the packets traverse the network [4]. The flow rules relayed by the controller are determined based on software applications designed and implemented by the network manager [1], [2]. The applications build flow rules using network topology information provided by the control plane [1]. Since the control plane is able to communicate with all the switches in the data plane, a complete topological view of the network is readily available [1]. The applications, controller, and switches exist at three logically or physically separated layers within the SDN framework according to the depiction in Figure 1.

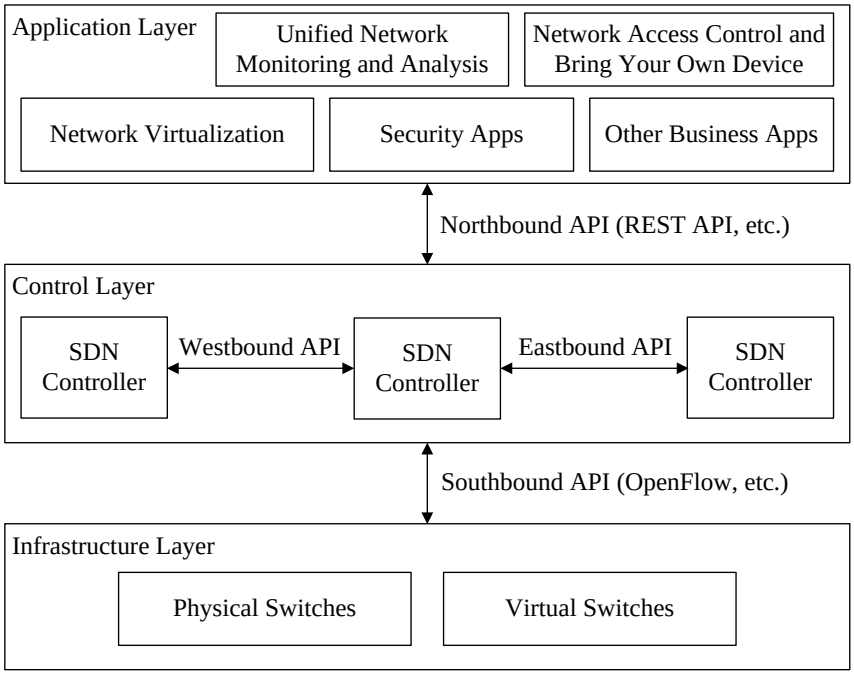

Figure 1. SDN Framework. Adapted from [2].

As evidenced by the framework displayed in Figure 1, layer-to-layer communication is facilitated by application programming interfaces (API). The control layer consists of a single controller or multiple controllers dependent on the setup of the network. If multiple controllers are used, separate APIs are also established between them for communication purposes. In this thesis, we are concerned with an SDN architecture consisting of physical switches at the infrastructure layer managed by a single SDN controller at the control layer. At the application layer, we employ software applications that implement port monitoring and topology view capabilities for security and MAC address forwarding for traffic routing.

The OpenFlow protocol represents a standardized southbound API used to facilitate communication between the control layer and the infrastructure layer [1], [2]. We employ the OpenFlow protocol in our SDN environment. In accordance with the OpenFlow protocol detailed in [12], messages traversing the bidirectional OpenFlow channel represented by the link between the controller and the switches are broken down into three types. The first message type is specified as a controller-to-switch message. It facilitates switch management and topology updates. Two critical examples of the

controller-to-switch message type are the Modify-State message and the Packet-out message. The Modify-State message allows the controller to manage the flow rules in the switches, and the Packet-out message enables the controller to present information to the switches for further forwarding. The second message type is specified as an asynchronous message. Asynchronous messages enable the switches to pass network topology and event information to the controller. A primary example of an asynchronous message type is the Packet-in message. Packet-in messages represent packets forwarded to the controller from the data plane switches [12]. The final message type is not a significant part of this thesis and is therefore not addressed. In this thesis, we rely on the Modify-State messages, Packet-out messages, and Packet-in messages to control traffic routing at the switches, obtain switch port updates, and build a complete view of the network topology. Additionally, the Packet-out and Packet-in messages are key components for the realization of the investigated MITM attack [4].

The northbound API, which facilitates communication between the application layer and control layer in accordance with Figure 1, is not defined by any current standards [2]. Specific details related to the northbound API are not of primary concern in this thesis.

B. CYBER EVENTS

In accordance with the stated objectives of this thesis, we seek to obtain traffic signals and anomalous conditions for multiple cyber events in order to compare them with the characteristics created by the MITM attack. An overview of the MITM attack studied in this thesis and the additional cyber events introduced for comparison are briefly presented in the following subsections.

1. MITM Attack

To develop a complete topology view at the SDN controller for input to the application layer, the controller must obtain link information between the switches in the data plane. The generally employed solution for link discovery by SDN controllers introduces the attack vector exploited by the investigated MITM attack [4].

Link discovery is facilitated for SDN controllers and switches communicating via the OpenFlow protocol by LLDP packets [13]. The process by which the LLDP packets are deployed and the link information is received at the controller is illustrated in Figure 2.

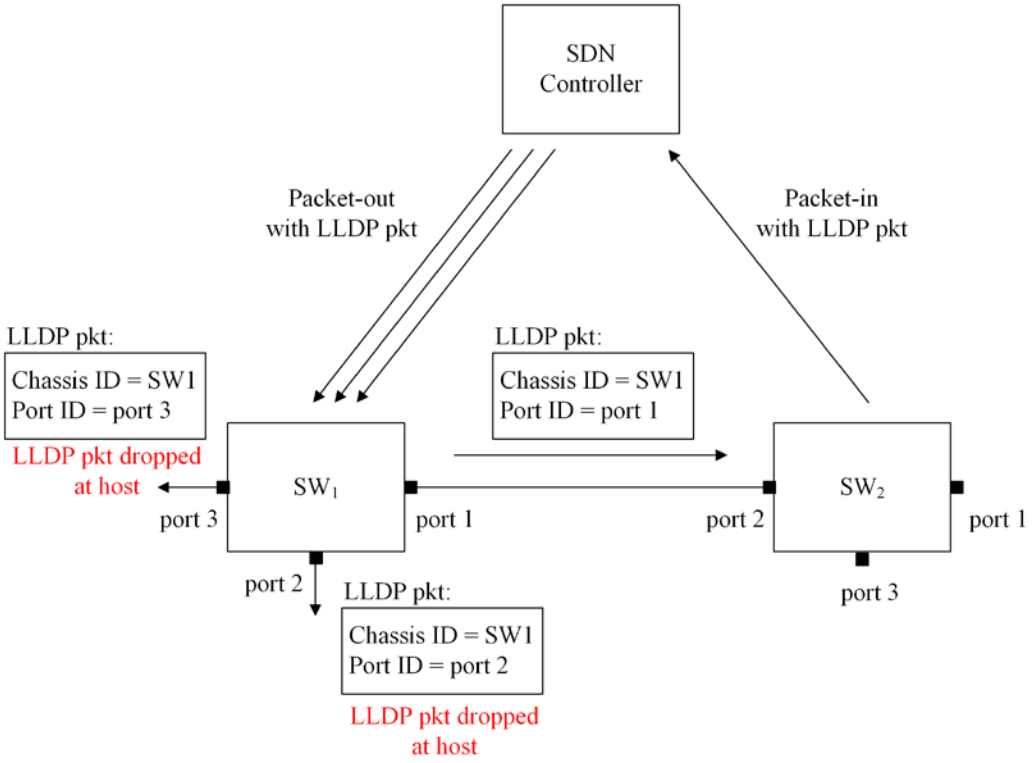

Figure 2. SDN LLDP Based Link Discovery Process. Adapted from [13].

In accordance with the link discovery process depicted in Figure 2 and detailed in [13], individual LLDP packets are sent from the controller using Packet-out messages at a continuous interval. The Packet-out messages are sent for each switch and their associated switch ports in the data plane. A Packet-out message contains a single LLDP packet populated with the switch identifier for which the Packet-out message was sent and the switch port number for further forwarding of the LLDP packet. If the forwarded LLDP packet is received by another switch, the receiving switch generates a Packet-in message containing the received port number and the received LLDP packet. The Packet-in message is then sent to the controller for creation of a directional link between the

forwarding switch port annotated in the LLDP packet and the received switch port annotated in the Packet-in message. If the forwarded LLDP packet is received by a host, it is dropped [13].

A sample Wireshark capture of a Packet-in message received at the controller during the link discovery process is displayed in Figure 3. The Packet-in message in Figure 3 coincides with the link discovery process displayed in Figure 2.

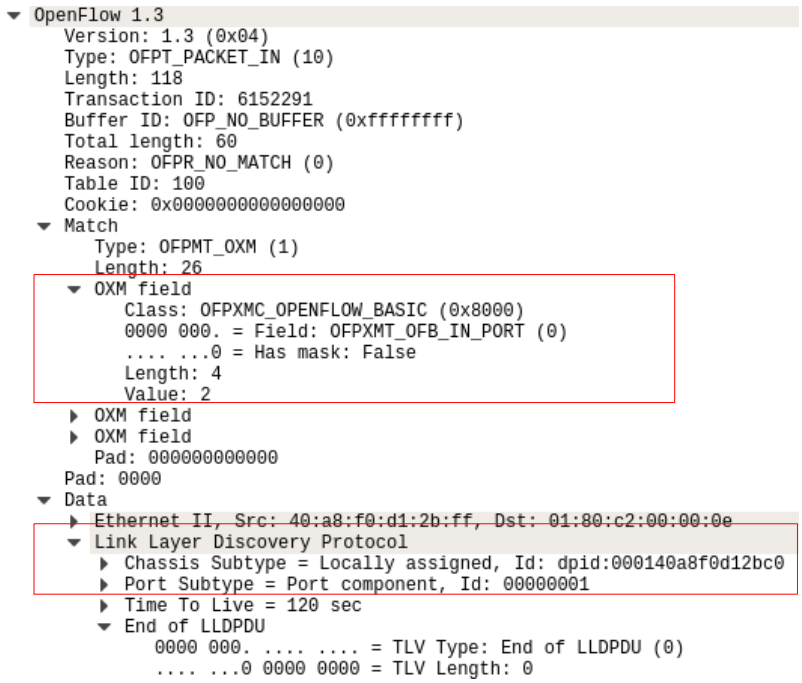

Figure 3. Sample Packet-in Message from the Link Discovery Process

The OpenFlow extensible match (OXM) field observable in Figure 3 displays the port number that the LLDP packet from switch SW_1 was received at switch SW_2. The LLDP packet sent from the controller and forwarded by switch SW_1 in Figure 2 is also displayed in the Packet-in message in Figure 3. The identifier for switch SW_1 and the forwarding port at switch SW_1 are clearly listed under the Link Layer Discovery Protocol heading. Based on the received Packet-in message displayed in Figure 3, the controller creates a link between port 1 of switch SW_1 and port 2 of switch SW_2.

The LLDP packets received at the network hosts open the door to the MITM attack investigated in this thesis. In accordance with the presented attack strategy in [4], LLDP packets received by two attacking hosts are relayed between each other over a maliciously constructed virtual or physical link instead of being dropped. The attack hosts represent previously compromised hosts in the SDN environment. A general example of the attack implementation with virtual tunnels is displayed in Figure 4.

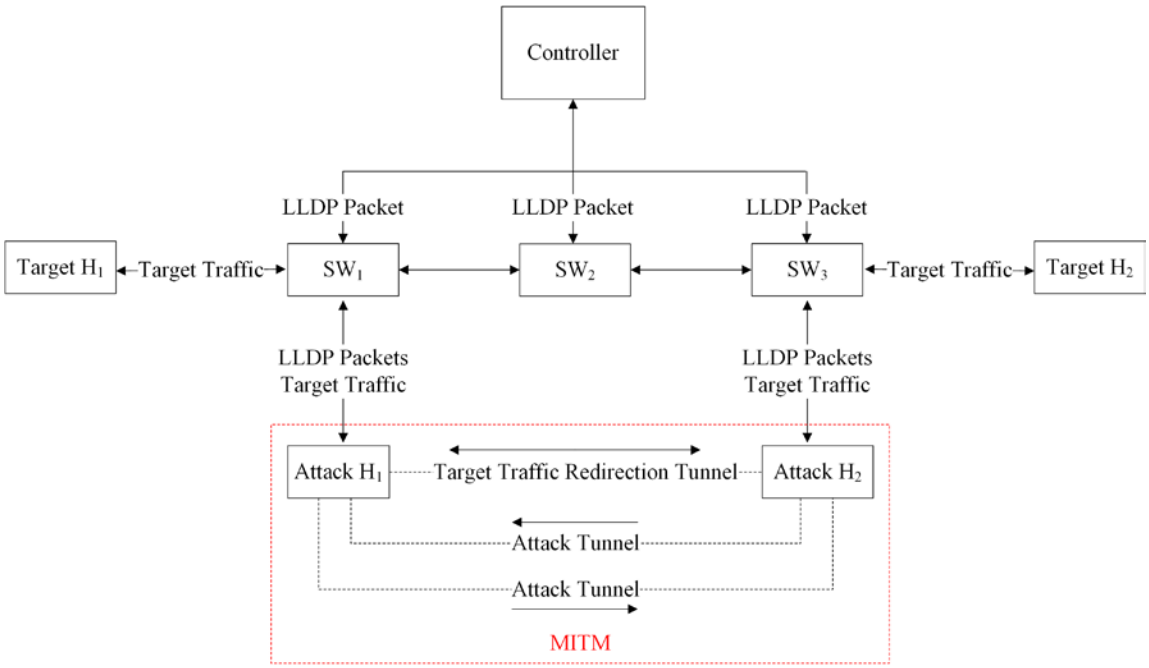

Figure 4. SDN MITM Attack Example. Adapted from [4].

Upon receiving the relayed LLDP packets, the attack hosts forward them back to their respective switches, and Packet-in messages for transfer of the LLDP packets back to the controller are created [4]. Since the LLDP packets were sent from the attack hosts to the switches, the received port numbers annotated in the Packet-in messages are associated with the attack hosts. This causes the controller to record a false link between the switches and through the attack hosts. After the controller updates switch-specific flow rules related to the false link, target host traffic is transmitted to the attack hosts and forwarded to the intended destination via a malicious virtual or physical link [4]. In Figure 4, the attack tunnels are used to relay the LLDP packets between the attack hosts

for creation of the false link. The target traffic redirection tunnel is used to transfer the target host traffic to its intended destination. The false link and the redirection of target traffic through the attack hosts create the conditions for the MITM attack [4].

In [4], direct tunnels are not established between the attack hosts as displayed in Figure 4. Instead, the LLDP packets and target host traffic are relayed between the attack hosts via an additional compromised host on the network. Implementing the attack in this thesis using the directly connected tunnels displayed in Figure 4 represents a simpler approach based on the overall attack strategy presented by [4].

2. DDOS Attack

A primary focus or strategy for the successful realization of denial of services is to overwhelm a target through bandwidth utilization [14]. A user datagram packet (UDP) flood represents a subset of DDoS attacks with this aim [15]. A typical UDP flood attack involves an attacker employing a network of controlled computers to produce a large volume of UDP packets destined for a target server [14], [15]. The flooding of the UDP packets hijacks the target server's available bandwidth and therefore cuts it off from the rest of the network [15]. We use the strategy employed by UDP flooding to achieve a DDoS attack against a target host in our SDN environment.

3. Network Congestion

Network congestion is represented by an observed reduction in throughput with increasing traffic volume [16]. It is typically experienced when the traffic volume overwhelms the capacity of the buffers in switching and routing devices [16]. The primary consequence of network congestion is congestive collapse where the throughput is reduced to levels that prevent communication between network devices [16]. In our SDN network, we seek to realize periods of congestive collapse between two hosts by introducing network congestion at an intermediate switch along a specified communication path.

C. WAVELETS

Wavelet analysis decomposes high and low frequency information in a signal into time and scale [17]. In wavelet analysis, a scaled wavelet with finite duration and zero mean is shifted across a signal to produce coefficient values representative of the wavelet's similarity to the processed portion of the signal [17]. The continuous wavelet transform coefficient values are calculated as given by [17]

$$C(\alpha,\beta) = \int_{-\infty}^{\infty} x(t)\psi(\alpha,\beta,t)dt\,, \tag{1}$$

where $C(\alpha,\beta)$ is the wavelet coefficient, α is the wavelet scale, β is the wavelet position, $x(t)$ is the signal, and $\psi(\alpha,\beta,t)$ is the wavelet. The position shifting and scaling of the wavelet for coefficient determination is displayed in Figure 5.

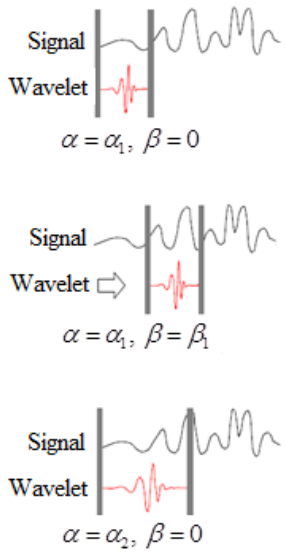

Figure 5. Shifting and Scaling the Wavelet Function. Adapted from [17].

In accordance with Figure 5, the wavelet at a particular scale $\alpha = \alpha_1$ is shifted across the signal for calculation of the coefficient values at different signal positions β. The wavelet scale is then decreased to $\alpha = \alpha_2$, stretching the duration of the wavelet, and additional coefficient values are obtained at the signal positions β. This process is

repeated for all scale values. Lower-scale values reveal abrupt, short-duration changes in the signal, while higher-scale values reveal longer-duration changes in the signal [17]. The scale value is limited by available computational capabilities [17].

In order to decompose the signals at faster speeds, the discrete wavelet transform (DWT) restricts the scaling and position values used in the analysis to powers of two [17]. High-pass and low-pass filters facilitate signal decomposition with the DWT and produce the coefficient values for an input signal at different scaling values. The low-pass filter generates approximation coefficients $a[n]$, which are associated with low-frequency signal components exposed by high-scale values. The high-pass filter generates detail coefficients $d[n]$ which are associated with high-frequency signal components exposed by low-scale values [17].

The DWT is capable of decomposing an input signal $x[n]$ at multiple resolution levels [17]. After obtaining the detail and approximation coefficients from an input signal $x[n]$, the filtering process is repeated with the generated approximation coefficients serving as the new input to the high and low-pass filters. The first set of approximation and detail coefficients obtained from the input of a signal $x[n]$ reflect the outputs of the DWT at level 1. The second set of approximation and detail coefficients obtained from the input of the level 1 approximation coefficients reflect the output of the DWT at level 2. Subsequent levels of approximation and detail coefficients are further obtained from the previous level's approximation coefficients [17]. The approximation and detail coefficients at each level of the DWT are given by [18]

$$a_j[n] = \sum_{i=-\infty}^{+\infty} h[i-2n]a_{j-1}[i] \tag{2}$$

and

$$d_j[n] = \sum_{i=-\infty}^{+\infty} g[i-2n]a_{j-1}[i]. \tag{3}$$

In Equations (2) and (3), $h[i-2n]$ represents the low-pass filter with down sampling, $g[i-2n]$ represents the high-pass filter with down sampling, and $j \geq 1$ represents the DWT level. Additionally, $a_0[n]$ represents the original signal $x[n]$.

The decomposed detail and approximation coefficients defined in Equations (2) and (3) do not match up with the time scale from the original signal $x[n]$ due to down sampling in the filtering process; therefore, they need to be reconstructed using similar filters with up sampling [17], [18]. Upon reconstruction, the detail and approximation coefficients correspond to the time scale of the original signal $x[n]$, and the low and high-frequency components extracted by the wavelet analysis are observable at the various levels. Furthermore, the reconstructed coefficient values can be combined to obtain the original signal [17]:

$$x[n] = A_j[n] + \sum_{k=1}^{j} D_k[n] \tag{4}$$

where $A[n]$ and $D[n]$ represent the reconstructed approximation coefficients $a[n]$ and detail coefficients $d[n]$, respectively, and $j \geq 1$ represents the DWT level.

Each successive DWT level provides a lower-resolution decomposition of the original signal $x[n]$ and is, therefore, able to reveal additional information [17]. In Figure 6, example reconstructed approximation and detail coefficient outputs at multiple levels of the DWT filtering process are displayed for a signal input $x[n]$. At level 3, we begin to observe more distinct information in the reconstructed detail coefficients of the transformed signal $x[n]$.

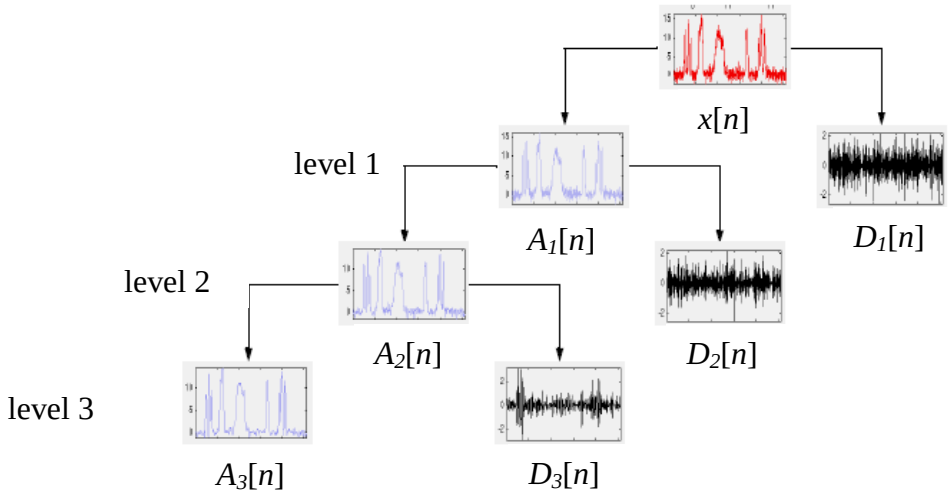

Figure 6. DWT Decomposition. Adapted from [17].

In this thesis, we decompose captured traffic signals with the DWT and use the reconstructed detail coefficients at multiple levels of the DWT to reveal high-frequency components in the signals [8]. Comparisons of the high frequency components with predetermined threshold values expose anomalous conditions in our traffic signals. As previously stated, the employment of wavelets to the anomaly detection problem is validated by research presented in [6]–[8].

In this chapter, we presented the necessary background information required to implement and test our proposed detection scheme. We started the chapter by introducing the components and basic operation of SDN. We then detailed the cyber events to be detected and classified. We ended the chapter with a description of wavelets and their applicability to anomaly detection.

THIS PAGE INTENTIONALLY LEFT BLANK

III. PROPOSED SCHEME FOR MITM ATTACK DETECTION AND CLASSIFICATION

The overall goal of this thesis is to detect and classify the SDN MITM attack proposed by [4] at the network layer based on its observed traffic signals. The proposed solution focuses on wavelet analysis for detection of anomalous traffic introduced by the attack and the localization of the detected anomalies for classification. Additional cyber events are introduced to the network in order to confirm the uniqueness of the detected anomalous traffic conditions for the MITM attack and validate the general applicability of the detection scheme.

We start this chapter by presenting the proposed scheme for detection and classification of cyber events on a SDN. We then detail the subcomponents of the proposed detection scheme.

A. PROPOSED SCHEME

As evidenced by previous research conducted in [6]–[8], a wavelet based analysis of traffic data for the detection of anomalies is valid. While [6]–[8] primarily expose anomalies relating to flash crowds, DoS attacks, port scans, and DDoS attacks, we extend their general approach to the detection of the SDN MITM attack presented in this thesis. The framework of our scheme is displayed in Figure 7 and is similar to the online anomaly detection schemes proposed in [7] and [8] with the addition of an anomaly classification phase. Individual details pertaining to the proposed detection scheme are detailed in the following sections.

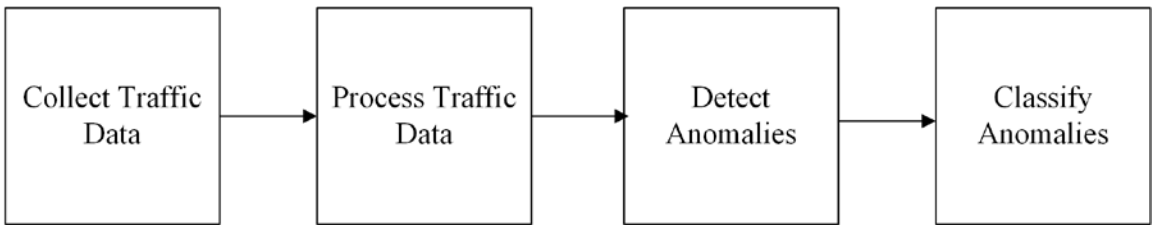

Figure 7. Proposed Anomaly Detection and Classification Scheme

B. COLLECTING TRAFFIC DATA

In order to obtain representations of the traffic signals related to the cyber events emulated in the network, byte, packet, and delay data are collected for host to host communications on the network. The network components involved in collection of byte, packet, and delay data within the general SDN environment employed in this thesis are displayed in Figure 8. The locations where byte and packet data are collected are outlined in blue. The locations where delay data is collected are outlined in green.

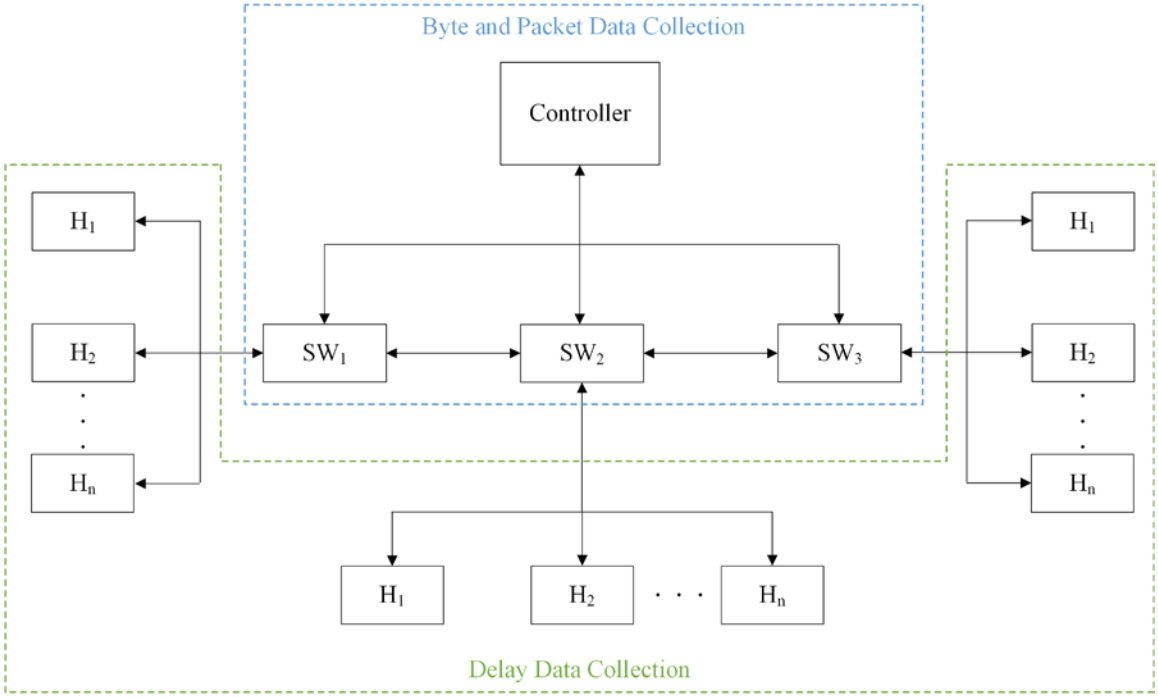

Figure 8. Data Collection in the SDN Environment

As depicted in Figure 8, byte and packet data are collected for both received and sent traffic at each active switch port on the network. The collection of this data is facilitated by an extension to the routing application running at the controller. The centralized nature of the controller and its inherent separation from the data plane makes this a simple and non-invasive task. The sampling rate is a variable parameter and can be adjusted by editing the controller application. The per-port traffic data is dumped to a separate file for generation of the corresponding cyber event traffic signals. The

generated traffic signals are then introduced as inputs to the data processing phase of the proposed detection scheme.

Traffic delay data between two communicating network hosts depicted in Figure 8 are captured using a software application at the receiving hosts. Collection of delay data is conducted passively and is focused on RTT between hosts in order to mitigate the requirement for synchronized network clocks critical to capturing one way delay [19]. The sampling rate is equal to that used for the collection of byte and packet data. The delay data are used to generate the delay signals for the cyber events. While the delay data are collected at the receiving hosts, for anomaly classification purposes it is referred to as if it were collected at the associated switch ports of the hosts. The delay signals serve as the third input to the data processing phase.

C. PROCESSING TRAFFIC DATA

The data inputs to the processing phase of our scheme are represented by the decomposition of each cyber event's traffic into captured byte, packet, and delay signals. The data points in the corresponding signals each represent a sampled value. The input signals $x[n]$ are processed separately and incrementally with the DWT based on the method presented in [8]. The specific wavelet used and its corresponding number of levels are selected by the network administrator.

Incremental processing with wavelets is conducted in accordance with a sliding window of length N_x samples, an even integer [8]. At each increment of the sliding window, the sample points of the input signal $x[n]$ that fall within the window are processed with the DWT. The detail coefficients produced by the DWT at each level are then reconstructed in order to expose the anomalous conditions within the timescale of the original signal $x[n]$. The sliding window moves across the input signal $x[n]$ until all signal values have been transformed with the DWT and reconstructed at each wavelet level [8].

The maximum number of wavelet levels l used to process $x[n]$ is limited by the sliding window length N_x, such that [8]

$$l = \log_2 N_x . \tag{5}$$

To account for edge effects as the sliding window moves across the signal inputs, an overlap parameter *v* is defined as

$$v = pN_x , \tag{6}$$

where *p* is the desired percentage of overlap between successive windows.

A signal input *x*[*n*] consisting of *N* samples, an even integer, can have a maximum number of sliding window increments N_w, given by

$$N_w = \left\lceil \frac{N}{N_x - v} \right\rceil . \tag{7}$$

From N_w, we obtain an increment vector $\vec{N}_1$ such that

$$\vec{N_1} = [2, 3, 4, ..., N_w - 1]. \tag{8}$$

With the increment vector $\vec{N}_1$ and maximum number of sliding window increments N_w, we define the right edge of the sliding window as

$$\vec{N_2} = \left[N_x, \vec{N_1}[k] N_x - \left(\vec{N_1}[k] - 1 \right) v, ..., \left(N_w - 2 \right) N_x \right] \tag{9}$$

for $k = 1, 2, 3, ..., N_w - 2$; therefore, based on the defined parameters, the sliding window for application of the DWT moving across input *x*[*n*] has intervals $\left[x\left[\vec{N_2}[j] - N_x + 1 \right], x\left[\vec{N_2}[j] \right] \right]$ for $j = 1, 2, 3, ..., N_w$. The sliding window method defined by Equations (6)-(9) is displayed in Figure 9.

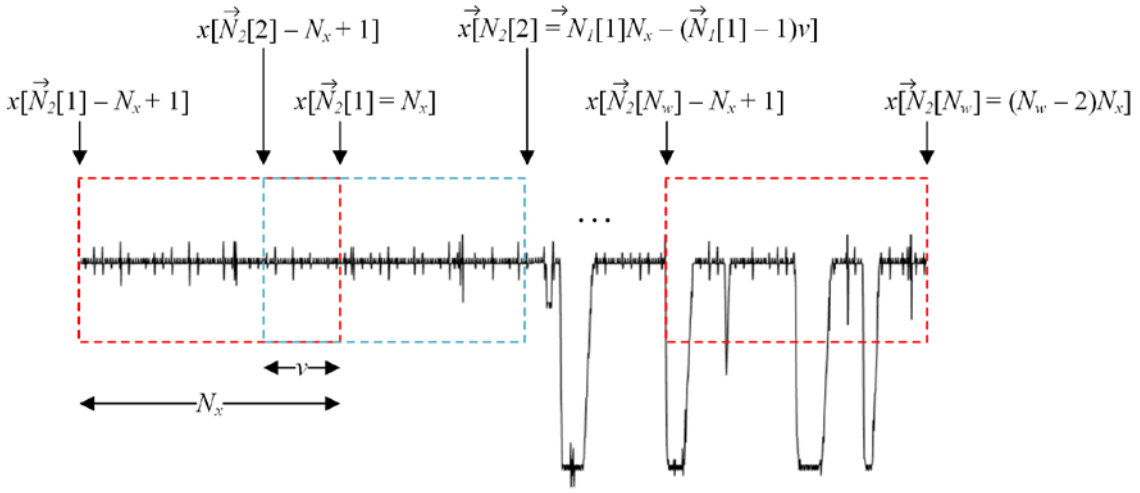

Figure 9. Sliding Window Method for Input Signal $x[n]$

In accordance with Figure 9, the DWT is applied to the sliding window intervals $\left[x\left[\overrightarrow{N_2}[j]-N_x+1\right], x\left[\overrightarrow{N_2}[j]\right]\right]$. The DWT produces detail coefficients representative of the input signal $x[n]$ within each sliding window interval, which are then reconstructed at their corresponding levels by forming a $N_x \times l$ matrix

$$Y_m = \begin{pmatrix} d_{1,1} & \cdots & d_{1,N_x} \\ \vdots & \ddots & \vdots \\ d_{l,1} & \cdots & d_{l,N_x} \end{pmatrix}, \tag{10}$$

where m denotes the numbered interval over which the DWT was applied. Each row in matrix Y_m therefore represents the transformation of $x[n]$ at the indicated sliding window interval and corresponding wavelet level. In total, for an $x[n]$ there are N_w matrix outputs Y_m each containing transformed and reconstructed representations of $x[n]$ at the specified interval and levels. These matrices are passed to the anomaly detection phase successively for further evaluation.

D. DETECTING ANOMALIES

The input Y_m to the detection phase of the proposed scheme consists of level specific reconstructed detail coefficients for the previously defined sliding window intervals over which the DWT was applied. The method of detecting anomalies for an

input Y_m is again adopted from [8] in which a detection window is used to determine initial anomalies at a specified level, and the reoccurrence of anomalies across multiple levels determines their validity. To represent the reconstructed detail coefficients at each wavelet level, the rows of a matrix input Y_m are arranged into separate row vectors $\vec{L}_y$ for $y = 1, 2, 3, ..., l$ as follows,

$$Y_m = \begin{bmatrix} \vec{L}_1 \\ \vec{L}_2 \\ \vdots \\ \vec{L}_l \end{bmatrix}. \tag{11}$$

The detection window searches for anomalies at each wavelet level by moving incrementally across the row vectors $\vec{L}_y$ of the reconstructed detail coefficients. The detection window length is N_d samples, where $N_d \leq N_x$ [8] and is an even integer. Counter to the application of the sliding window for the DWT, the starting position of the detection window increments by one sample at a time [8]. The detection window intervals are defined by $\left[\vec{L}_y[k], \vec{L}_y[k + N_d - 1]\right]$ for $k = 1, 2, 3, ..., N_x - N_d$. At each interval, the mean absolute deviation M is calculated as [20]

$$M = \frac{1}{N_d} \sum_{i=k}^{k+N_d-1} \left| \vec{L}_y[i] - \overline{\left[\vec{L}_y[k], \vec{L}_y[k + N_d - 1]\right]} \right| \tag{12}$$

and compared with a predefined threshold value γ_{lev} [7]. If the threshold value γ_{lev} is exceeded, an initial anomaly is logged for the corresponding wavelet level [8]. The interval of the anomaly is defined within the original signal $x[n]$ by $\left[x\left[\overrightarrow{N_2}[j] - N_x + 1 + k\right], x\left[\overrightarrow{N_2}[j] - N_x + 1 + k + N_d\right]\right]$.

Once all the anomalies are detected for each wavelet level within a matrix Y_m, anomalies appearing in multiple levels are logged [8]. If the number of occurrences for each anomaly across all the wavelet levels meets or exceeds the predefined threshold γ_{rep}, they are reported as valid anomalies [8]. Using the windowed approach for the DWT and detection window in concert with the two phase threshold comparison is shown to

minimize false alarms [8]. The decision process for anomaly detection as previously described is further illustrated by the flowchart in Figure 10.

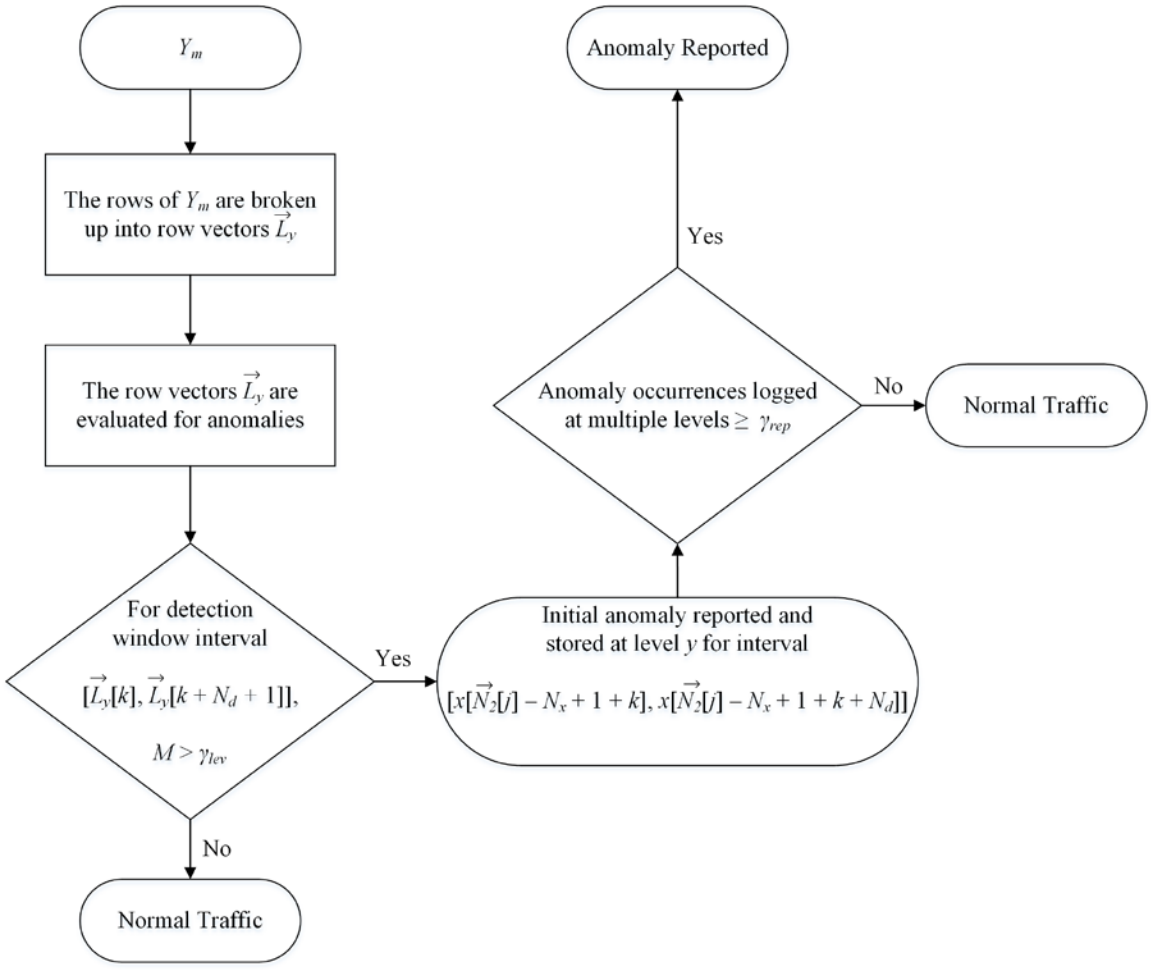

Figure 10. Anomaly Detection Flowchart

It is assumed that a specific host switch port will not experience simultaneous cyber events. To mitigate subsequent or overlapping anomalies being reported for the same cyber event conditions, the intervals between reported anomalies for an $x[n]$ must be separated by a specified number of samples defined as a back-off period P_b, given by

$$P_b = qN_x, \tag{13}$$

where q is the percentage of back off. Two detected anomalies that are not separated by the specified back-off period are considered a single anomaly with an inclusive interval.

The threshold values γ_{lev} used to detect initial anomalies within the wavelet levels are determined using statistical analysis [8]. Statistical methods for anomaly detection base the determination of anomalies on their inability to conform to an assumed stochastic model [21]. When a Gaussian distribution is selected as the assumed stochastic model for the data under consideration, a threshold of 3σ from the mean represents the boundary for conforming observations, where σ is the standard deviation [21]. Data observations outside of the 3σ boundary are logged as anomalies [21].

In [8], traffic data under normal operation is fit to a Gaussian distribution by sampling enough traffic to meet the conditions of the central limit theorem, analyzing the data with the DWT, and rebuilding a representation of the data by combining the reconstructed detail coefficients from the specific levels of the DWT. The basis for rebuilding a representation of the original traffic data by combining the reconstructed detail coefficients is supported by Equation (4). In this case, the reconstructed approximation coefficients A_j are set to zero, and only the reconstructed detail coefficients are considered. Setting the reconstructed approximation coefficients to zero and combining the reconstructed detail coefficients results in a representation of the original traffic signal that is Gaussian and justifies the use of statistical analysis for the assignment of threshold values [8]. Upon validating that the representation of the normal traffic data satisfies the Gaussian requirement, the authors establish threshold values for anomaly detection based on the standard deviations of the reconstructed detail coefficients at each DWT level [8].

We apply the same overall method for assignment of our threshold values γ_{lev}. Normal traffic signals without injection of the previously specified cyber events are collected at the switch ports of two communicating hosts and transformed with the DWT. The normal traffic signals consist of received bytes, sent bytes, received packets, sent packets, and delay. The detail coefficients associated with each normal traffic signal are then reconstructed, combined, and fit to a Gaussian distribution to validate the use of statistical analysis. Upon validating the Gaussian requirement, we assign threshold values γ_{lev} at each switch port of the communicating hosts based on the standard deviations of the represented normal traffic signals obtained by combining the reconstructed detail

coefficients. As a result, we assign five threshold values γ_{lev} at a single port; each threshold value is associated with a collected normal traffic signal at the port. Contrary to [8], we do not assign separate threshold values for each DWT level of a normal traffic signal in an effort to reduce false positives. Further efforts to reduce false positives are implemented by considering threshold values consisting of 3σ and 4σ in Chapter IV. Both threshold values were shown to be acceptable in [8].

The threshold value γ_{rep} represents the desired number of repeat occurrences of an anomaly across employed wavelet levels [8]. The value is used to tune the detection method to obtain the best results possible and reduce false positives. The selected value is not based on mathematical observations.

E. CLASSIFYING ANOMALIES

The inputs to the anomaly classification phase of our proposed scheme consist of the confirmed anomalies output by the detection phase. The flowchart for determination of the MITM attack explored in this thesis is displayed in Figure 11. The flowchart was constructed based on initial observations of traffic characteristics during emulations of the specific cyber events of interest. While it was not the primary objective of this thesis, in addition to the MITM attack, we are able to individually classify the DDoS and congestion cyber events. The method for cyber event classification based on the localization of anomalous conditions in network traffic signals was validated in [11].

Based on Figure 11, anomalous traffic reported at each port on a switch is considered for classification of the introduced cyber events. The DDoS and congestion cyber events are classified based on the detection of anomalies in the delay signal and the byte or packet signals at a single port. The congestion cyber event is further differentiated from the DDoS attack by a lack of detected anomalies in the sent byte or packet signals. The MITM cyber event is classified based on the presence of anomalous traffic at two different ports on a switch. Anomalous conditions at each port will lead to a conditional classification of the MITM attack via the different paths represented in the flowchart. The switch port associated with a target host of the MITM attack will experience anomalies in the measured delay signal but not the measured byte or packet signals. The switch port

associated with an attacking host will experience anomalies in the measured received or sent byte or packet signals but not the measured delay signal. The indication of a possible MITM attack at both ports confirms the classification of the cyber event. While the MITM attack involves attacking and target hosts on two different switches, anomalies only need to be observed at one of the switches to successfully classify the cyber event.

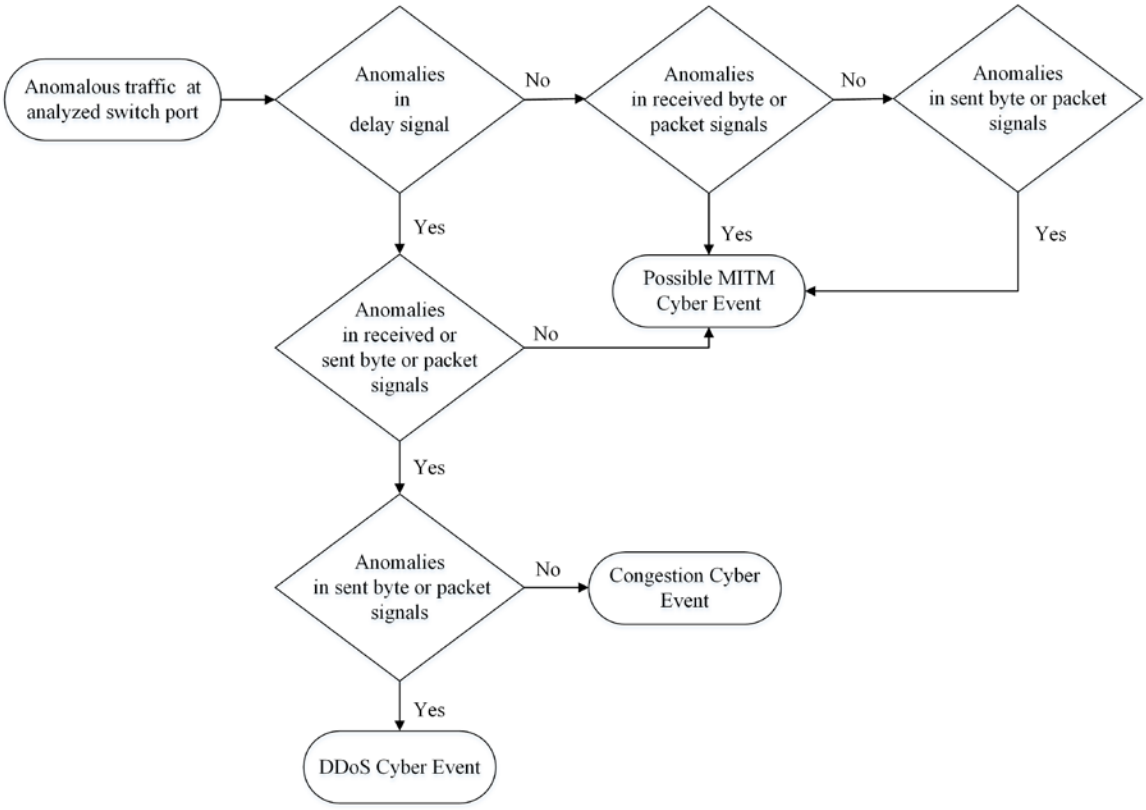

Figure 11. Cyber Event Classification Flowchart

In this chapter, we presented the proposed scheme for detection and classification of the selected cyber events of interest and detailed its individual components. The detection scheme employs wavelet and statistical analysis for the exposure and reporting of anomalous conditions related to these cyber events. Further classification of the cyber events is based on a decision flowchart developed using initial observations of traffic characteristics from the emulated cyber events.

IV. RESULTS

In order to validate the proposed detection scheme presented in Chapter III, we emulated a MITM attack [4], a DDoS attack, and network congestion in our physical SDN environment. Using collected traffic signals from the corresponding cyber events, we processed the signals using wavelet analysis and compared the analyzed signals with predetermined threshold values for anomaly detection. The uniqueness of the reported anomalies for the individual cyber events served as logic based indicators for the classification flowchart presented in Chapter III.

The results obtained in this thesis are presented in the following sections. First, we present a detailed description of our physical SDN environment and the software applications used to generate and capture necessary traffic signals. Next, we detail the implementation of the cyber events in the physical SDN environment and present the captured traffic signals for those cyber events. Lastly, we present the detailed results obtained from the processing, detection, and classification phases of our proposed scheme. All captured byte signals for the respective tests were converted to bits in order to more clearly reveal the anomalous conditions in the associated plots. Yet, they are still referred to as byte signals throughout this thesis.

A. SDN ENVIRONMENT AND TRAFFIC CAPTURES

The physical SDN environment used for the collection of traffic signals related to the cyber events presented in Chapter II is displayed in Figure 13. The control plane consisted of a Ryu controller running the Ryu applications *simple_monitor13.py* and *gui_topology.py* [22]. The *simple_monitor13.py* application created the initial flow rules on the network for host-to-host communication and collected the byte and packet data at the switch ports for the generation of traffic signals [23]. For each cyber event introduced to the network, the switch ports were sampled at one second intervals, and the data was separated based on received and sent traffic. The *gui_topology.py* application provided a topology display of the data plane at the controller [24]. To obtain the links to build the topology view at the controller, the *gui_topology.py* application relies on *switches.py*

which employs the vulnerable LLDP packets necessary to realize the SDN MITM attack [4], [22]. We confirmed the successful implementation of the SDN MITM attack upon observing the creation of a false link in the topology view. The operating system employed on the controller was Ubuntu 14.04 LTS.

The data plane consisted of three Hewlett Packard (HP) E3800 switches connected to the controller via a NETGEAR GS516T switch. There were 16 hosts connected to the data plane switches. The hosts consisted of Raspberry Pi's running the Ubuntu 16.04 LTS operating system and personal computers running the Ubuntu 14.04 LTS operating system. The controller and HP switches were configured to communicate via the OpenFlow version 1.3 protocol.

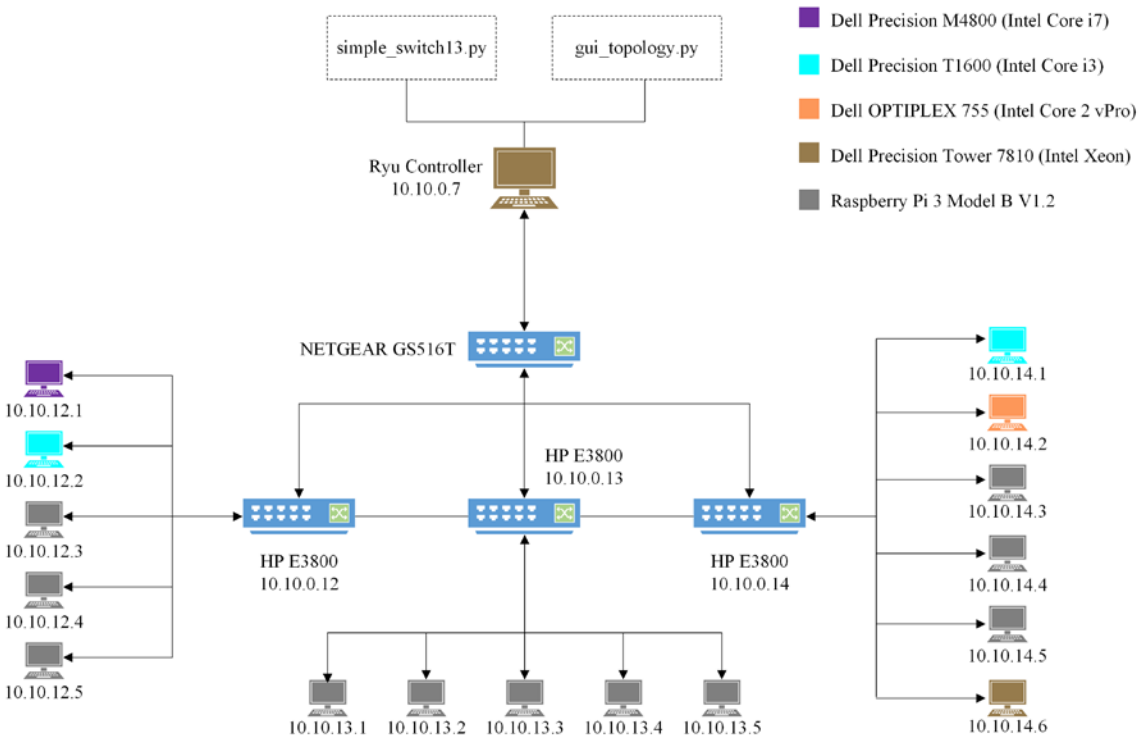

Figure 13. Physical SDN Environment

Traffic was generated within the network test environment using the *iperf* tool [25]. An *iperf* server session was created on each receiving host prior to running the cyber event tests. To send the traffic, *iperf* client sessions were started on each sending

host remotely from host 10.10.14.6 using a bourne-again shell script. All generated traffic consisted of UDP streams. Each test was conducted for over an hour in order to obtain 3,600 samples for each traffic signal. The traffic sampling period was one second.

The RTT data for generation of the delay signals was calculated using synthetic packet-pairs (SPP) as described in [19]. The SPP tool enables captured packets between two communicating hosts to be analyzed for accurate RTT calculations based on introduced time stamps at the capture nodes [19]. The tool can be configured to operate in multiple modes, both online and offline, dependent on a network manager's desires and tolerance for overhead [19]. In our mode of operation, we captured packet traces for two communicating hosts and combined the packet traces offline to be analyzed by the SPP tool. The SPP tool does not provide the option to set the sampling period; therefore, the final data was converted to represent a one-second sampling period.

Traffic signals were only generated from collected traffic data at host ports necessary to detect and classify the cyber events considered. These ports were assigned based on initial observations of anomalous conditions introduced by the cyber events.

B. CYBER EVENT TRAFFIC SIGNALS

The cyber events were emulated in the SDN environment during separate tests. Each test consisted of a period of normal traffic prior to the introduction of a cyber event in order to validate our method of detection. As stated previously, each test was conducted for more than an hour in order to obtain a minimum of 3,600 samples of the byte, packet, and delay traffic signals.

1. MITM Attack

We implemented the MITM attack presented by [4] in our physical SDN environment using a combination of virtual tunnels, Linux bridges, and virtual interfaces. The attack details are illustrated in Figure 14. The attack hosts are highlighted in red and represent previously compromised hosts on the network. The target hosts are highlighted in green.

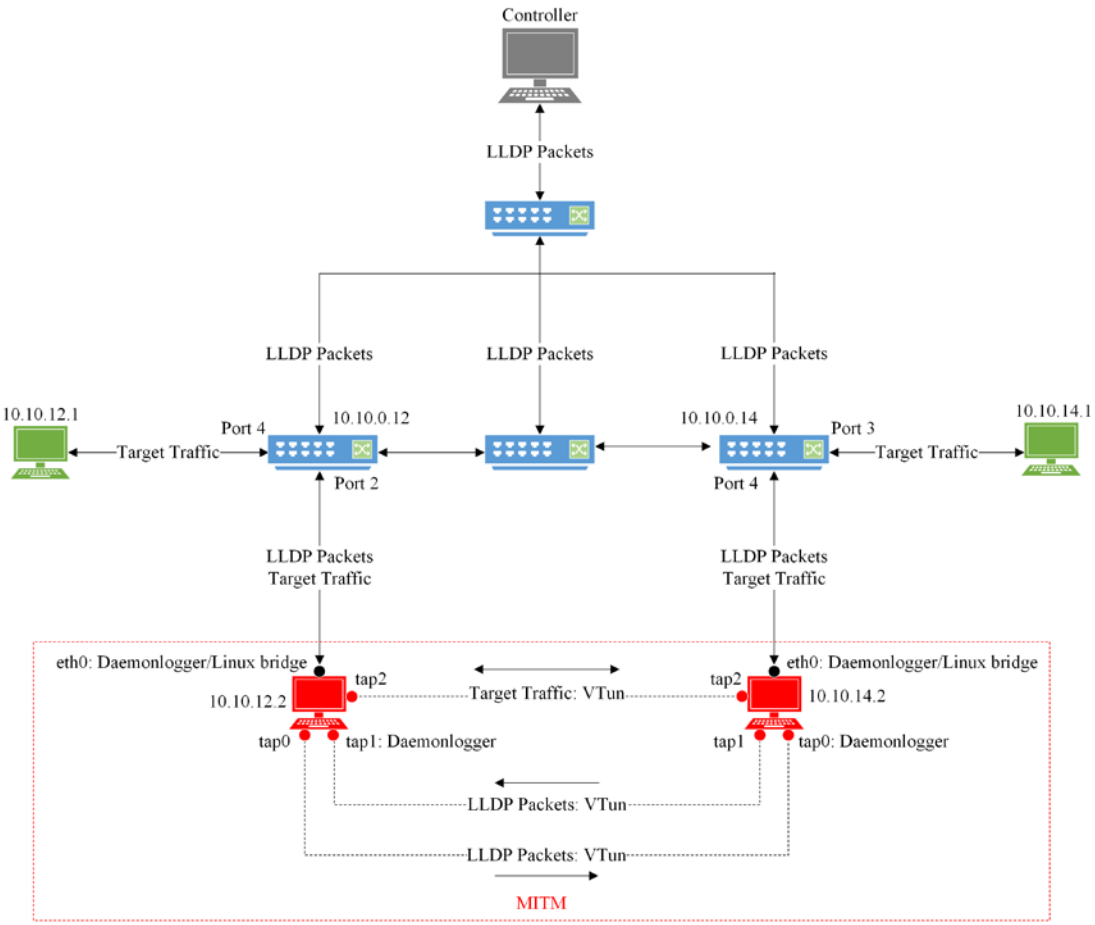

Figure 14. MITM Attack Implementation

In accordance with the MITM attack strategy described in Chapter II and further illustrated in Figure 14, the attack hosts 10.10.12.2 and 10.10.14.2 captured LLDP packets from switches 10.10.0.12 and 10.10.0.14 at their physical interfaces, eth0. These captured LLDP packets were then relayed to their respective virtual interfaces, tap0 and tap1, for transport over the virtual tunnels linking the attack hosts together. The *Daemonlogger* tool [26] was used to capture and relay the LLDP packets between the physical and virtual interfaces. The tunnels were set up with the VTun (short for Virtual Tunnel) protocol for Linux [27]. When the attack hosts received each other's relayed LLDP packets at the terminating ends of the virtual tunnels, the LLDP packets were forwarded to their physical interfaces, eth0, using *Daemonlogger* and sent to their associated switches. The switches then populated Packet-in messages with their

respective information and the relayed LLDP packets for transmission back to the controller. Upon receipt of the Packet-in messages with the relayed LLDP packets at the controller, the topological view was updated with a false bidirectional link and new flow rules were manually installed to account for the false link information. A third tunnel was then set up between the attack hosts in order to deliver the traffic redirected by the updated flow rules to the target hosts 10.10.12.1 and 10.10.14.1.

To maintain the MITM attack, the LLDP packets were continuously relayed between the attack hosts. If the LLDP relay was interrupted, the false link was no longer recognizable at the controller, and the attack ended. Utilizing separate tunnels to relay the LLDP packets between the attack hosts ensured the stability of the attack. The topological view at the controller representing our successful implementation of the MITM attack is displayed in Figure 15. The false link is represented by the connection between switches 10.10.0.12 and 10.10.0.14.

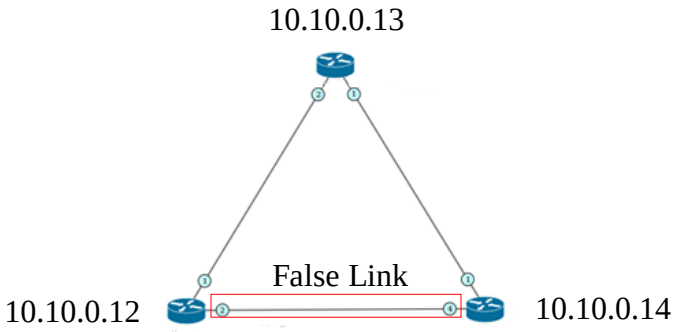

Figure 15. Ryu Topology View of MITM Attack

To capture the traffic signals associated with the attack, received byte and packet data was collected at the port for the attack host 10.10.12.2 on switch 10.10.0.12. Sent byte and packet data was ignored due to its similarities to the received byte and packet data. Delay data was collected between the target hosts 10.10.12.1 and 10.10.14.1. To start the test, separate bidirectional UDP streams were established between the target hosts and the attack hosts. Additional UDP streams were established between other network hosts in order to provide background traffic. All traffic flowed through the network via the actual links for a period of about 20 minutes. This period represented

normal traffic conditions. At about the 20-minute mark, the MITM attack was initiated, and the bidirectional UDP stream between the target hosts was redirected through the attack hosts. All other UDP streams continued to flow through the network normally. The traffic signals associated with the attack are displayed in Figure 16. The signals represent the received byte and packet data captured at the switch port of the attack host 10.10.12.2 and the delay data captured at the target host 10.10.12.1. Each signal reflects the anomalous conditions introduced by the attack.

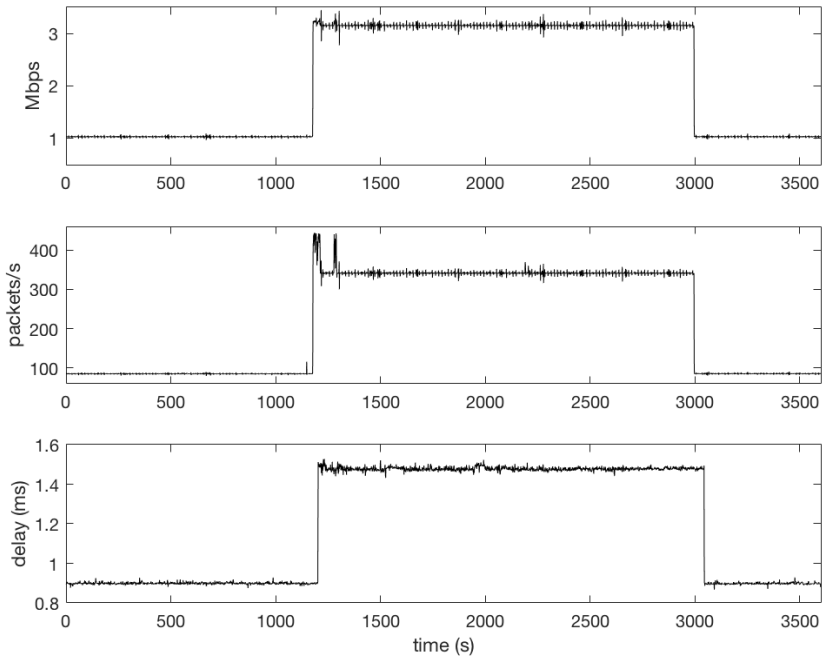

Figure 16. MITM Attack Traffic Signals Captured at Hosts 10.10.12.1 (Bottom) and 10.10.12.2 (Top and Middle)

As evidenced by the displayed signals in Figure 16, the attack host experiences a sharp increase in its received byte and packet signals upon implementation of the attack. This is caused by the redirection of the target host traffic through the attack host. The attack host 10.10.12.2 receives all of the traffic sent by the target host 10.10.12.1 in addition to the normal traffic being sent to it from the other attack host 10.10.14.2. The delay signal between the communicating target hosts also experiences a sharp increase when the attack is implemented. This is caused by redirection of the traffic and the added

processing delay imposed at the attack hosts. The traffic must travel to the attack hosts before being sent over the virtual link to its intended destinations.

The collected traffic signals for the MITM attack displayed in Figure 16 are representative of the anomalous conditions observed during the test. All other traffic signals or delay signals associated with the ports and hosts of switch 10.10.0.12 did not reveal significant anomalous conditions for classification of the attack; therefore, they were ignored.

2. DDOS Attack

We implemented the DDoS attack in our physical SDN environment using nine attack hosts. The DDoS attack type was a UDP flood. The attack scenario is illustrated in Figure 17. The green host represents the target, and the red hosts represent the attackers.

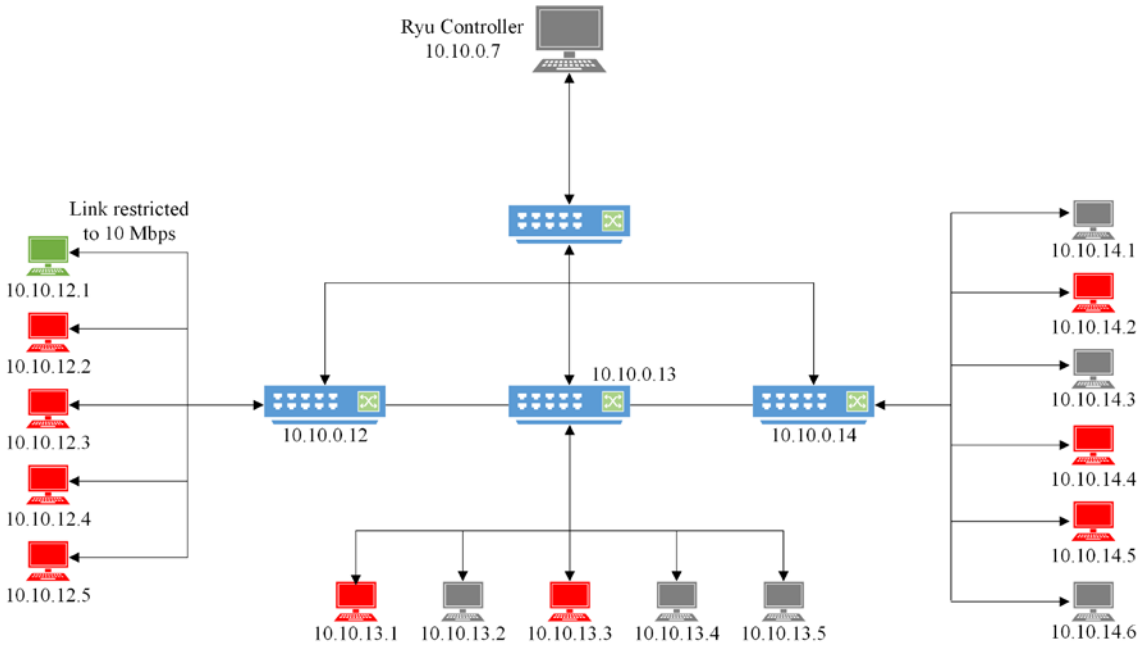

Figure 17. DDoS Attack Implementation

To start the test, a normal bidirectional UDP stream was established between hosts 10.10.12.1 and 10.10.14.1. The attack target was host 10.10.12.1. To realize the attack conditions at the target host port, UDP streams were established from the nine

attacking hosts to the target host. The DDoS attack was introduced after 20 minutes of normal traffic and lasted for 10 minutes. The overall test was conducted for more than an hour in order to collect 3,600 samples representing normal and attack traffic. The switch port for the target host 10.10.12.1 was restricted to a data rate of 10 Mbps in order to achieve the attack conditions with the limited resources available in the test environment.

Based on the collected delay data and traffic statistics at the target host port, traffic signals were generated to highlight the effects of the DDoS attack. The byte and packet signals are representative of received and sent traffic statistics at the target host port. The delay signal is representative of experienced RTTs between the communicating hosts 10.10.12.1 and 10.10.14.1. The traffic signals generated from the captured byte and packet data are displayed in Figure 18. The traffic signal generated from the delay data is displayed in Figure 19. Anomalous conditions caused by the introduction of the DDoS attack are clearly visible in these figures.

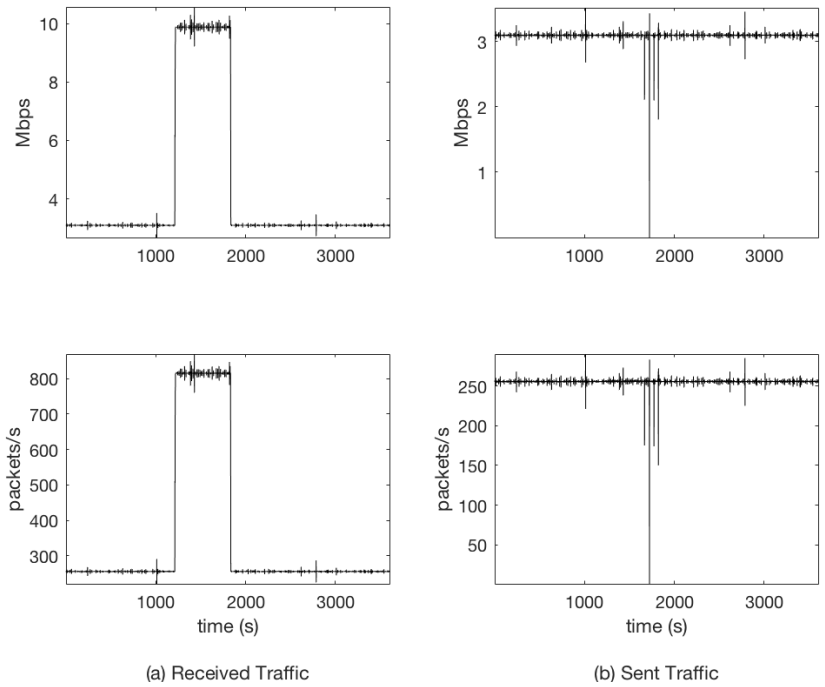

Figure 18. DDoS Attack Byte and Packet Signals Captured at Host 10.10.12.1

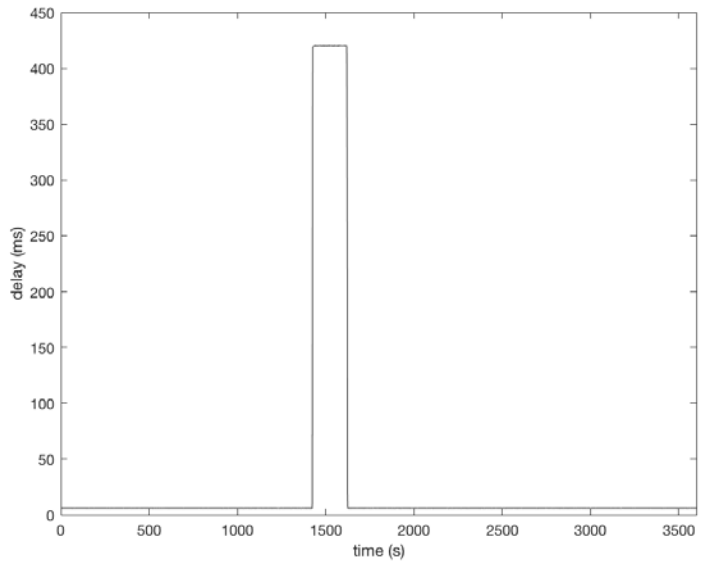

Figure 19. DDoS Attack Delay Signal Captured at Host 10.10.12.1

In Figure 18, the received byte signal increases from about 3 Mbps under normal traffic conditions to 10 Mbps upon initiation of the attack. This is a result of the attacking hosts flooding the target host with UDP traffic. Similarly, the received packet signal jumps from about 250 packets per second to 800 packets per second when the attack is introduced.

The impacts of the attack are also observable in the sent traffic signals of Figure 18 and the delay signal of Figure 19. The sent traffic signals begin to fall to zero, and communication between hosts 10.10.12.1 and 10.10.14.1 starts to break down as the attacking hosts overtake the available bandwidth at the target host. In Figure 19, the delay signal increases from less than 1 ms under normal traffic conditions to over 400 ms under attack conditions. Overall, the combination of anomalous conditions provides valuable information required to differentiate the DDoS attack from the other emulated cyber events.

3. Congestion

We introduced congestion to the network at switch 10.10.0.13. The process by which the congestion was implemented is displayed in Figure 20. The switch and link

highlighted in red represent the congested components. For the test, traffic was generated on the network by establishing UDP streams between paired hosts with *iperf*. As displayed in Figure 20, the data rate was limited to 10 Mbps at port 2 of switch 10.10.0.13 in order to force congested conditions for traffic traversing the link between switches 10.10.0.13 and 10.10.0.14.

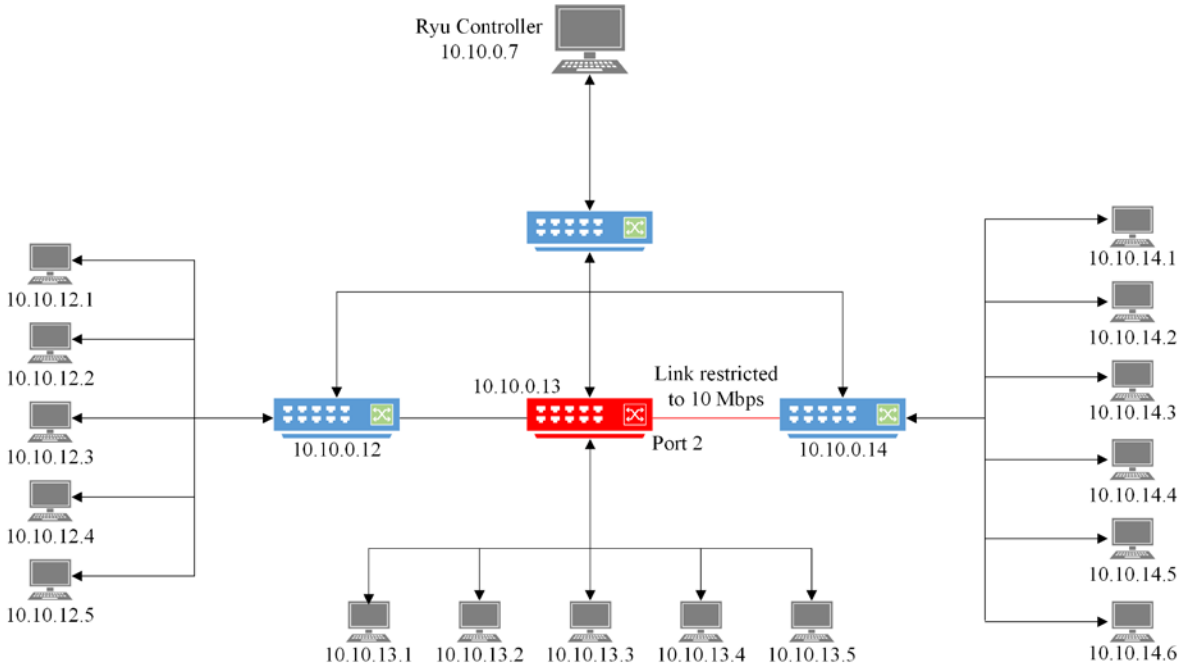

Figure 20. Introduction of Network Congestion

During the test, we collected traffic data on switch 10.10.0.12 at the port associated with host 10.10.12.1. Delay data was collected between communicating hosts 10.10.12.1 and 10.10.14.1. The test was conducted for over an hour in order to obtain 3,600 samples of the traffic data. The test was started by only generating traffic between hosts 10.10.12.1 and 10.10.14.1 in order to obtain normal traffic data. After a period of about 20 minutes, the rest of the communicating host pairs began to generate traffic. We were able to observe the effects of congestion in the network as the traffic volume overwhelmed the 10 Mbps capacity at port 2 of switch 10.10.0.13. The traffic signals displaying the effects of the introduced congestion at the switch port for host 10.10.12.1 are displayed in Figure 21.

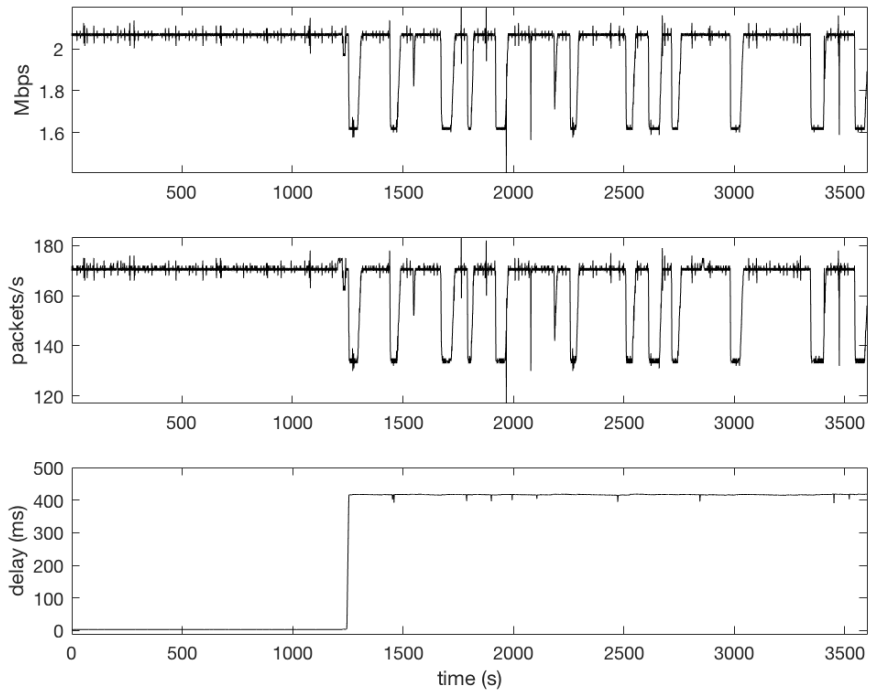

Figure 21. Congestion Traffic Signals Captured at Host 10.10.12.1

In Figure 21, the traffic signals are representative of received traffic data at the switch port for host 10.10.12.1 and delay data experienced between host 10.10.12.1 and 10.10.14.1. The effects of congestion are clearly observable. The byte and packet rates begin to drop and the delay increases at the 20-minute mark when congestion is initiated in the network.

The sent byte and packet signals collected at the switch port for host 10.10.12.1 are displayed in Figure 22. The signals do not reveal any observable anomalous conditions resulting from congestion. The data rates are relatively constant. This is consistent with expectations since the effects of congestion on sent traffic from host 10.10.12.1 are only introduced at switch 10.10.0.13. We collected sent traffic data at the switch port associated with host 10.10.12.1 before it traversed the congested switch link. The lack of anomalous conditions in the sent byte or packet signals for the congestion cyber event is a key differentiator from the DDoS attack in the anomaly classification flowchart displayed in Figure 11.

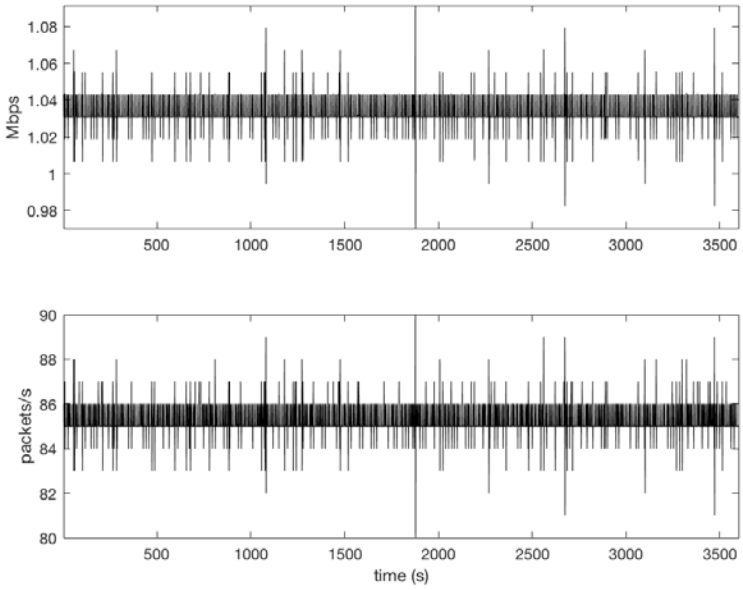

Figure 22. Sent Traffic Signals Captured at Host 10.10.12.1 for Congestion

C. DETECTION OF THE CYBER EVENTS

After collecting the traffic signals containing observable anomalous conditions for the emulated cyber events, our next step was to autonomously detect and report the anomalies. To accomplish this task, the traffic signals presented in the previous sections were passed to the data processing and anomaly detection phases of the scheme displayed in Figure 7. The data processing and anomaly detection phases were implemented in MATLAB.

1. Determination of the Threshold Values

In order to determine the threshold values γ_{lev} for anomaly detection, normal traffic was generated in the physical SDN environment, and data signals were collected for wavelet processing. Based on the previously presented traffic signals, anomalous conditions for the introduced cyber events were observed at the switch ports for hosts 10.10.12.1 and 10.10.12.2; therefore, bidirectional UDP streams were established between hosts 10.10.12.1 and 10.10.14.1 and hosts 10.10.12.2 and 10.10.14.2 for collection of delay and port traffic data under normal network conditions. The UDP

streams were maintained for a time duration exceeding an hour in order to obtain traffic signals consisting of 3,600 samples. The baseline traffic signals related to byte and packet data were obtained for received and sent traffic at the associated switch ports to hosts 10.10.12.1 and 10.10.12.2. The baseline traffic signal related to delay data was obtained for the communicating host pair 10.10.12.1 and 10.10.14.1. The traffic signals for baseline traffic conditions at hosts 10.10.12.1 and 10.10.12.2 are displayed in Figures 23–25. Contrary to the traffic signals displayed in Figures 16, 18, 19, and 21, there are no anomalous conditions observable in the traffic signals displayed in Figures 23–25. The byte and packet signals in Figures 23 and 25 maintain a relatively consistent data rate of 1 Mbps and packet rate of 86 packets per second, respectively. The delay signal in Figure 24 stays fairly constant at 0.91 ms. These baseline traffic signals are used for determining the threshold values γ_{lev} at the switch ports for hosts 10.10.12.1 and 10.10.12.2.

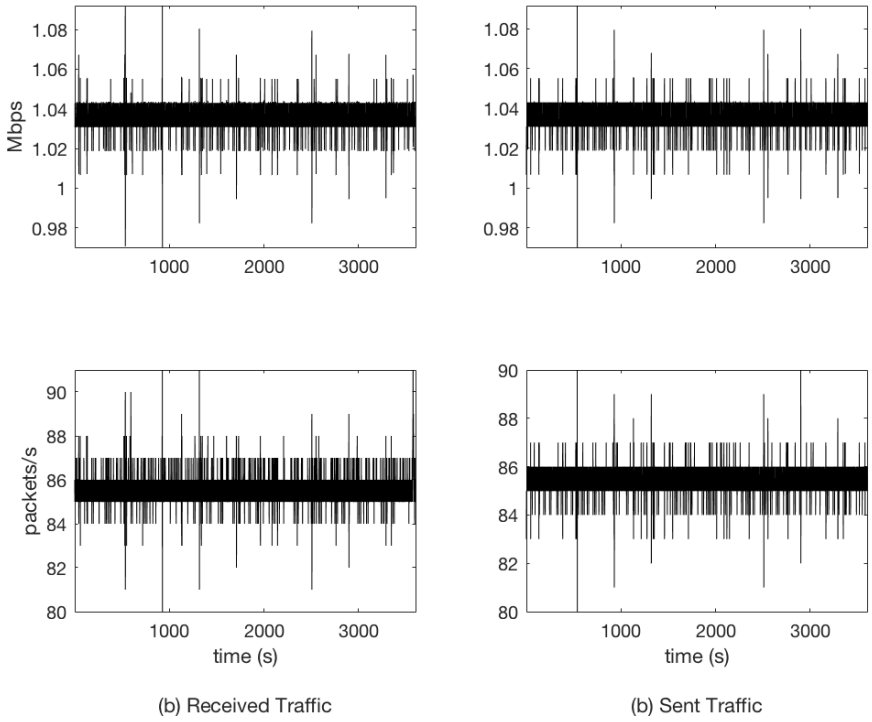

Figure 23. Normal Traffic Signals Captured at Host 10.10.12.1

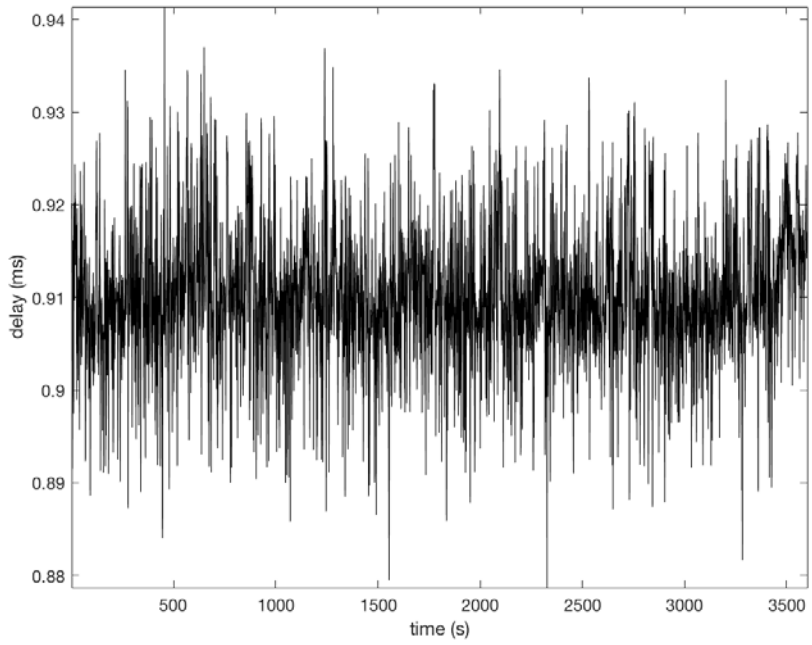

Figure 24. Normal Delay Signal Captured at Host 10.10.12.1

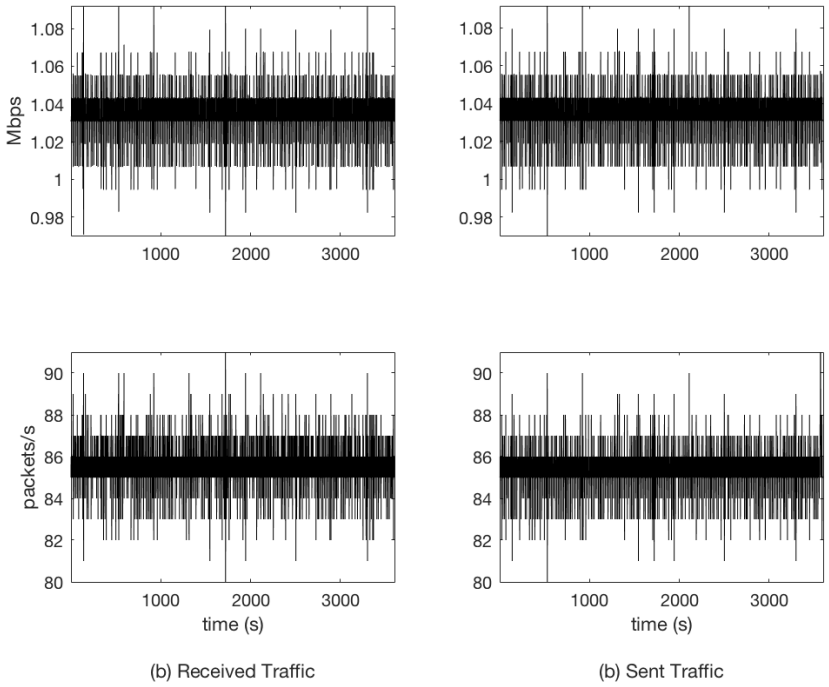

Figure 25. Normal Traffic Signals Captured at Host 10.10.12.2

In this thesis, we chose to use the Daubechies-1 wavelet to process traffic signals for anomaly detection and determination of the threshold values associated with normal traffic conditions. The Daubechies wavelet family was shown to successfully expose anomalous traffic in [8]. In [8], the Daubechies-6 wavelet was used. We found that the Daubechies-1 wavelet produced better results for anomaly detection in our collected traffic signals.

To obtain the threshold values for anomaly detection at the switch ports for hosts 10.10.12.1 and 10.10.12.2, the traffic signals displayed in Figures 23–25 were decomposed at six wavelet levels. The detail coefficients at each level were then combined to obtain representations of each signal. As discussed in Chapter III, Section D, the assignment of threshold values based on statistical analysis is reliant on the transformed representations of the normal traffic signals being Gaussian [8], [21]. The histograms of the transformed signal representations are displayed in Figures 26–28.

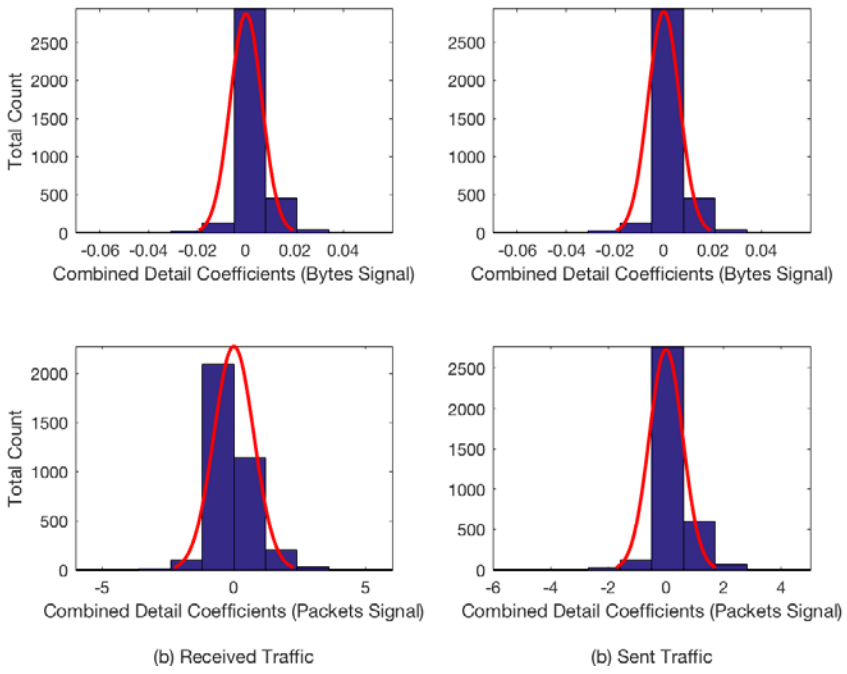

Figure 26. Histograms of Transformed Signal Representations for Normal Traffic at Host 10.10.12.1

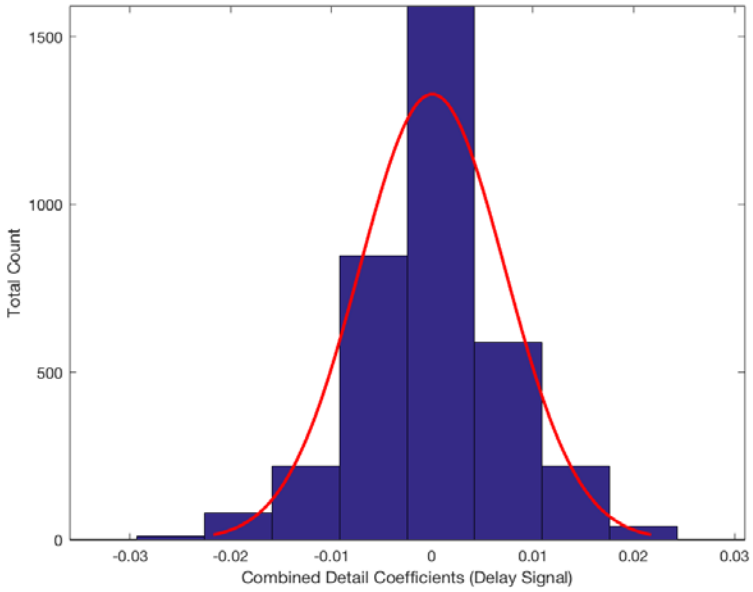

Figure 27. Histogram of Transformed Signal Representation for Normal Delay Data at Host 10.10.12.1

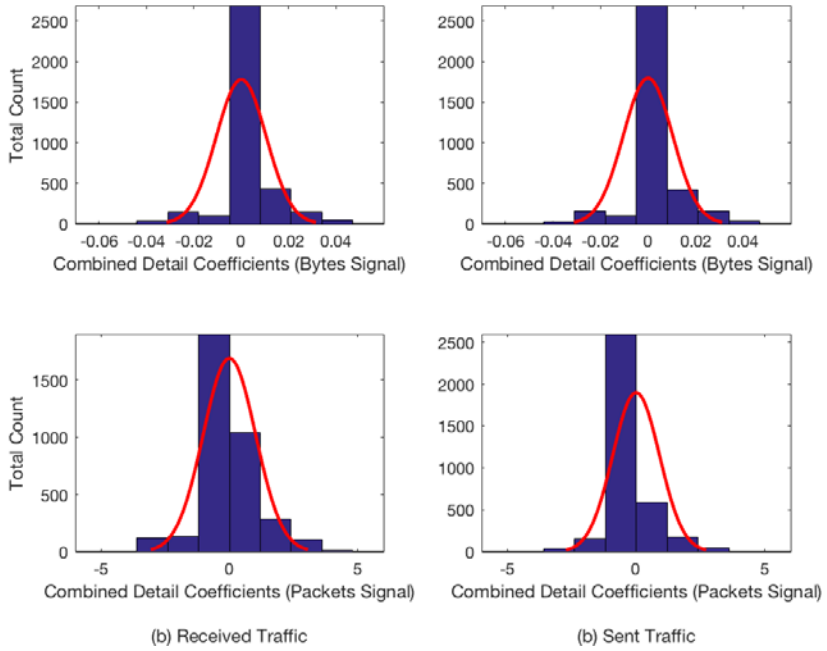

Figure 28. Histograms of Transformed Signal Representations for Normal Traffic at Host 10.10.12.2

The histograms resemble the characteristics of a Gaussian distribution; therefore, the threshold values γ_{lev} for each host switch port were assigned based on the standard deviations of their transformed signal representations. The threshold values γ_{lev} for anomaly detection at the host switch ports impacted by the introduced cyber events are presented in Table 1. Threshold values were only obtained for traffic signals that displayed observable anomalous conditions during each cyber event.

Table 1. Threshold Values at 3σ and 4σ of the Transformed Normal Traffic Signals

		Host			
		10.10.12.1		10.10.12.2	
	Multiple of Standard Deviation	3σ	4σ	3σ	4σ
γ_{lev}	Received Traffic Signal (Bytes)	0.0194	0.0259	0.0315	0.0420
	Sent Traffic Signal (Bytes)	0.0193	0.0257	0.0312	0.0416
	Received Traffic Signal (Packets)	2.2668	3.0225	3.0583	4.0777
	Sent Traffic Signal (Packets)	1.7335	2.3113	2.7241	3.6322
	Delay Signal	0.0217	0.0290	N/A	N/A

In [8], upon validating that the combined detail coefficients for a transformed normal traffic signal fit to a Gaussian distribution, separate threshold values were assigned based on the standard deviations of the detail coefficients at each reconstructed wavelet level. We found that using a single threshold value based on the standard deviations of the combined detail coefficients resulted in fewer false positives when applied to the reconstructed wavelet levels of anomalous traffic. We presume that this was due to limited variability observed in our baseline traffic signals. Overall, the use of a single threshold value for each transformed signal representation provided more conservative requirements for determination of anomalies, thereby reducing the overall number of false positives.

2. Determination of the Detection Parameters

Several parameters for the data processing and detection methods proposed in Chapter III were examined in order to determine the values that most accurately exposed the anomalous conditions observable in the captured traffic signals. Other parameters

were held constant throughout the processing and detection phases. The parameter values used are shown in Table 2.

Table 2. Detection Parameters

Parameter	Value
N_x	Varied
l	6
p	10%
N	3,600
N_d	Varied
γ_{lev}	Varied
γ_{rep}	4
q	20%

Each captured traffic signal was decomposed with the Daubechies-1 wavelet and reconstructed at six levels. The reconstructed signals were then compared with the threshold values for anomaly detection in accordance with the flowchart presented in Figure 10. The associated false positives and false negatives for different wavelet window sizes N_x, detection window sizes N_d, and threshold values γ_{lev} used in the data processing and anomaly detection phases of the detection scheme are displayed in Tables 3–5. For each case, the number of false positives and false negatives are annotated in the columns labeled P and N, respectively. The false positives and false negatives were determined based on observable anomalous conditions in the previously presented traffic signals. If an anomaly was reported for an interval in the captured traffic signals that appeared normal, it was labeled as a false positive. If a clearly observable anomalous condition for the captured traffic signals was not reported by the detection method, it was labeled as a false negative.

Table 3. MITM Attack Detection Results for Varied Parameter Values

			Wavelet Window Size (N_x)																							
			200								300								600							
			Detection Window Size (N_d)								Detection Window Size (N_d)								Detection Window Size (N_d)							
			10		25		50		100		10		25		50		100		10		25		50		100	
			P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N
γ_{lev}	3σ	Received Bytes	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	2
		Received Packets	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		Delay	0	0	0	0	0	0	0	2	0	0	0	0	0	1	0	2	0	0	0	0	0	0	0	2
	4σ	Received Bytes	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	2
		Received Packets	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
		Delay	0	0	0	0	0	1	0	2	0	0	0	0	0	2	0	2	0	0	0	0	0	1	0	2

Table 4. DDoS Attack Detection Results for Varied Parameter Values

			Wavelet Window Size (N_x)																							
			200								300								600							
			Detection Window Size (N_d)								Detection Window Size (N_d)								Detection Window Size (N_d)							
			10		25		50		100		10		25		50		100		10		25		50		100	
			P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N
γ_{lev}	3σ	Bytes Received	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0
		Packets Received	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		Bytes Sent	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
		Packets Sent	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
		Delay	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	4σ	Bytes Received	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		Packets Received	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		Bytes Sent	1	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	0
		Packets Sent	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	0
		Delay	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5. Congestion Detection Results for Varied Parameter Values

			Wavelet Window Size (N_x)																							
			200								300								600							
			Detection Window Size (N_d)								Detection Window Size (N_d)								Detection Window Size (N_d)							
			10		25		50		100		10		25		50		100		10		25		50		100	
			P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N
γ_{lev}	3σ	Received Bytes	0	0	0	1	0	7	0	8	0	1	0	1	0	3	0	5	0	0	0	0	0	5	0	7
		Received Packets	0	3	0	7	0	11	0	14	0	1	0	3	0	10	0	14	0	2	0	5	0	9	0	11
		Sent Bytes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		Sent Packets	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		Delay	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	4σ	Received Bytes	0	2	0	6	0	10	0	14	0	1	0	4	0	7	0	14	0	1	0	5	0	9	0	11
		Received Packets	0	9	0	10	0	15	0	16	0	7	0	11	0	15	0	16	0	8	0	11	0	13	0	16
		Sent Bytes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		Sent Packets	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		Delay	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Based on the results displayed in Tables 3–5, a wavelet window size $N_x = 300$, a detection window size $N_d = 10$, and the level threshold value $\gamma_{lev} = 3\sigma$ produced the best detection results. The false positives and negatives associated with the parameter values are highlighted in the tables by the red boxes. The parameter values referenced above reduced the total number of observed false positives to one and the total number of observed false negatives to two across all of the cyber event traffic signals. The two false negatives were observed in the received byte and packet traffic signals of the congestion cyber event and represent the same anomalous traffic interval.

3. Reporting Anomalous Traffic Conditions

The detected and reported anomalous traffic conditions for the various cyber events emulated in the SDN test environment are displayed in Figures 29–35. As stated previously, the traffic conditions for all cyber event tests were sampled at one-second intervals. The vertical red lines in all the figures represent the beginning and end times of the detected anomalies in the analyzed traffic signals. The captured traffic signals and their reconstructed detail coefficients at wavelet levels 1–6 are depicted in the figures. The anomalous conditions visible in the traffic signals are emphasized at each wavelet level by spikes in the detail coefficients. The reported anomalies that met the threshold conditions are displayed in the traffic signal and all its reconstructed wavelet levels. The levels at which they were detected are not indicated. As evidenced by the results highlighted in Tables 3–5, similar anomalies were reported in the byte and packet signals using the final parameter values; therefore, the anomalous conditions reported in the packet signals of each cyber event are not displayed.

a. MITM Attack

The anomalous traffic conditions based on the byte signal for received traffic at the attack host 10.10.12.2 during the MITM attack are observed in Figure 29. The anomalous traffic conditions based on the delay signal captured at host 10.10.12.1 are illustrated in Figure 30.

In Figure 29, anomalies were detected and reported at 1,168 to 1,188 seconds and 2,984 to 3,012 seconds. The first anomaly represents the sharp jump in received traffic as the MITM attack was initiated and the attack host 10.10.12.2 started receiving all the traffic sent by the target host 10.10.12.1. The second anomaly represents a sharp drop in the traffic signal after the attack was ended and the network returned to forwarding traffic in accordance with its physical connections. All other traffic in the collected signal appears normal and is free of any anomalies. The same anomalies at the beginning and end of the MITM attack are observed in the delay signal. There were not any false positives or negatives reported by the detection method.

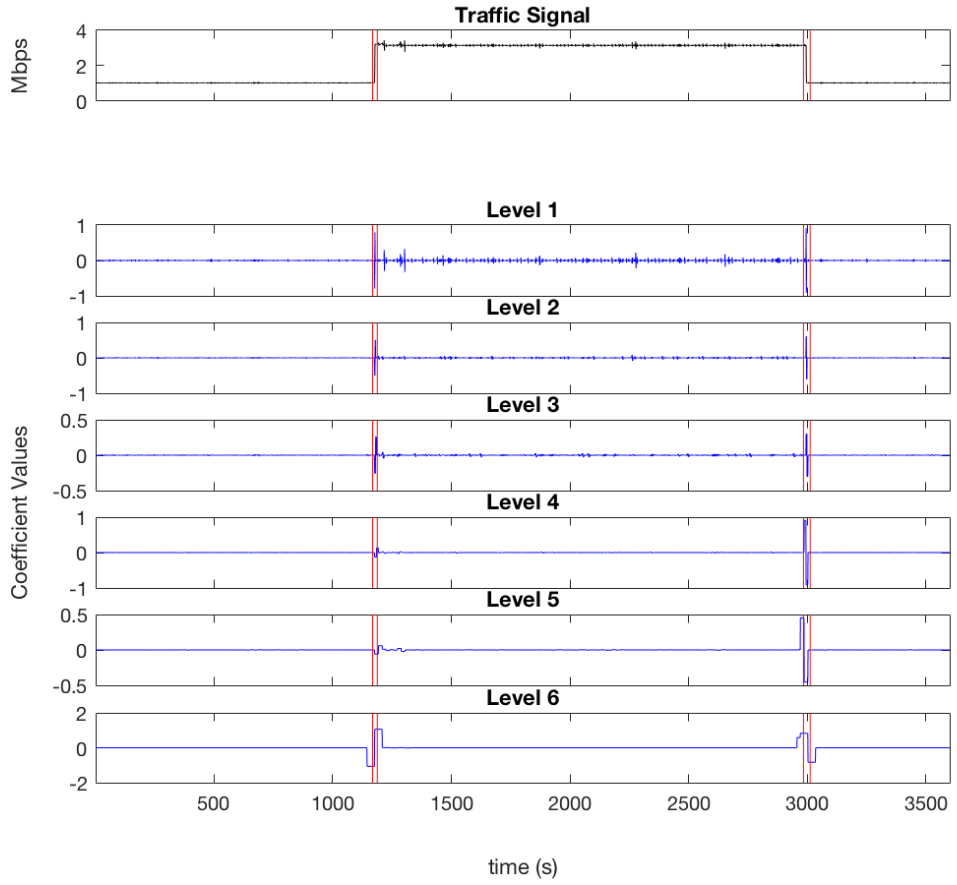

Figure 29. MITM Attack Detection in Received Byte Signal at Host 10.10.12.2

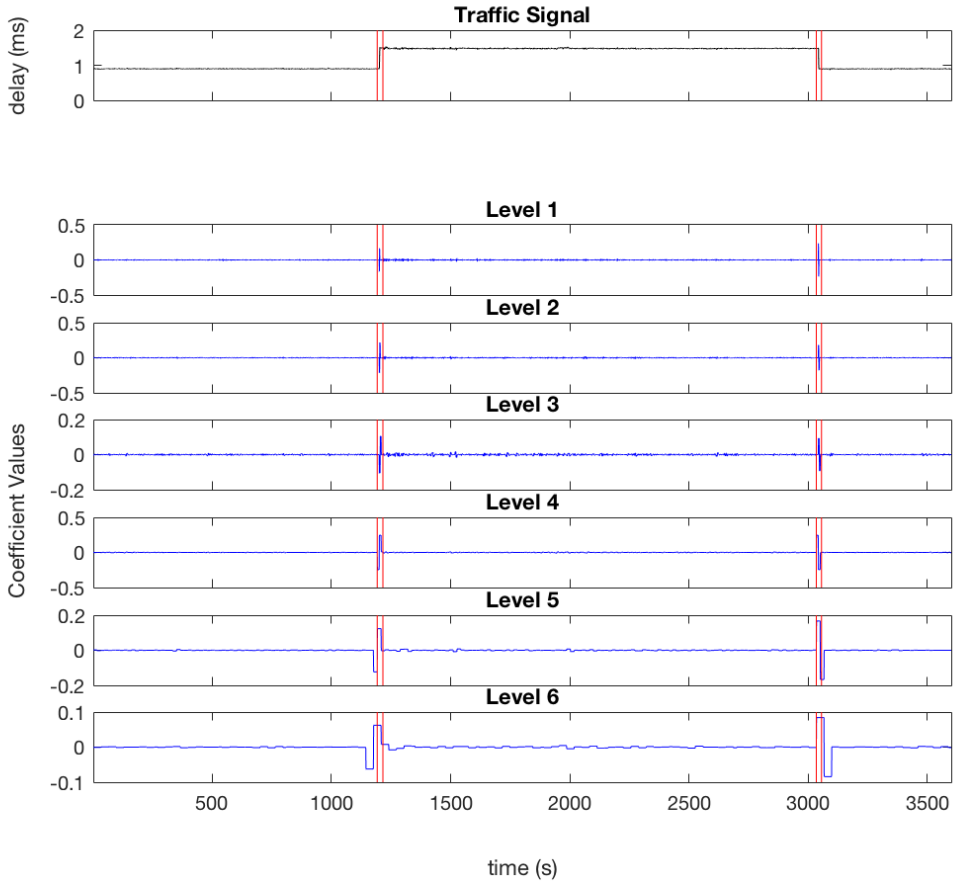

Figure 30. MITM Attack Detection in Delay Signal at Host 10.10.12.1

b. DDoS Attack

The anomalous traffic conditions based on the byte signal for received traffic at the target host 10.10.12.1 during the DDoS attack are revealed in Figure 31. The data rates rose almost instantaneously as the attack was initiated, and the target host was flooded with UDP traffic from the nine attacking hosts. The data rates returned to normal when the attack was ended. The beginning and end of the DDoS attack are accurately reported by the anomaly detection method in the received byte signal. The anomaly intervals reported in the received byte signal are at 1,200 to 1,230 seconds and 1,806 to 1,846 seconds. One false positive is reported in the received byte signal of Figure 31 at 1,385 to 1,410 seconds. The false positive is not a major concern because it occurs during the DDoS attack; therefore, it does not indicate a separate attack. As evidenced by the

results highlighted in Table 4, the false positive was not reported in the packet signal for received traffic. There were no false negatives reported.

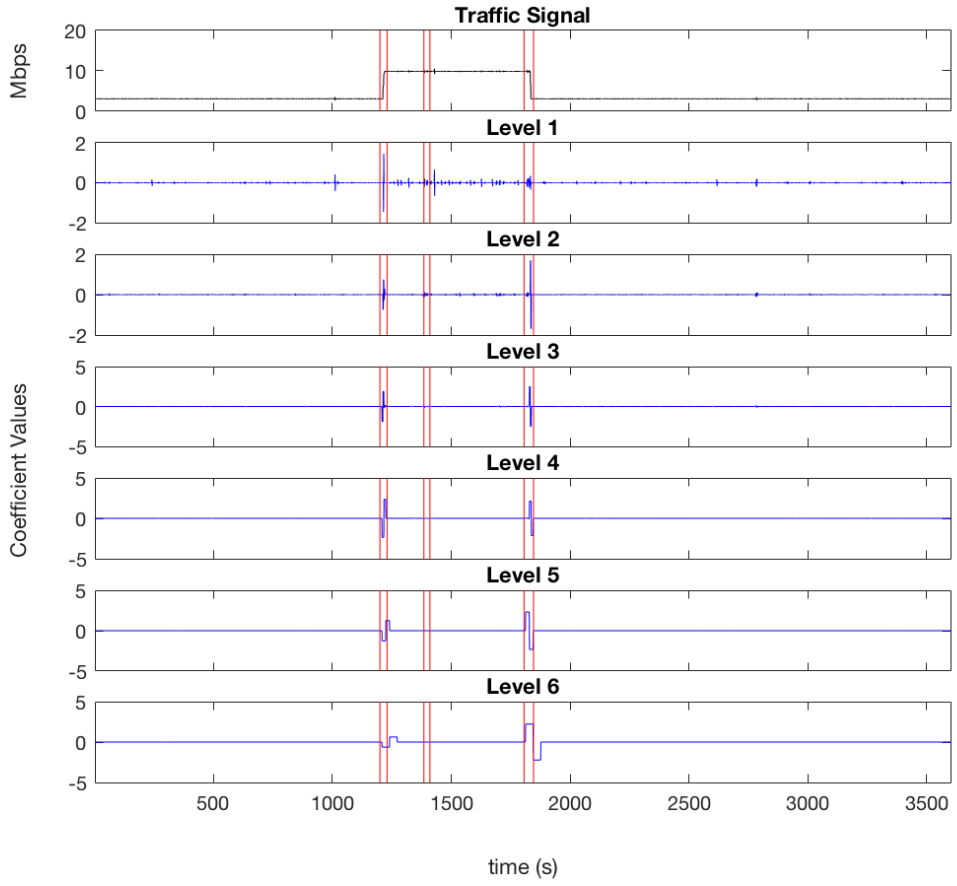

Figure 31. DDoS Attack Detection in Received Byte Signal at Host 10.10.12.1

The anomalous traffic conditions based on the byte signal for traffic sent by the target host of the attack are revealed in Figure 32. The target host experienced degraded communication with its intended recipient 10.10.14.1 due to the DDoS attack during the interval from 1,658 to 1,836 seconds. The anomalous conditions are clearly visible in the traffic signals and are accurately reported by the detection method. The detection method did not report any false positives or negatives in the captured sent traffic signal.

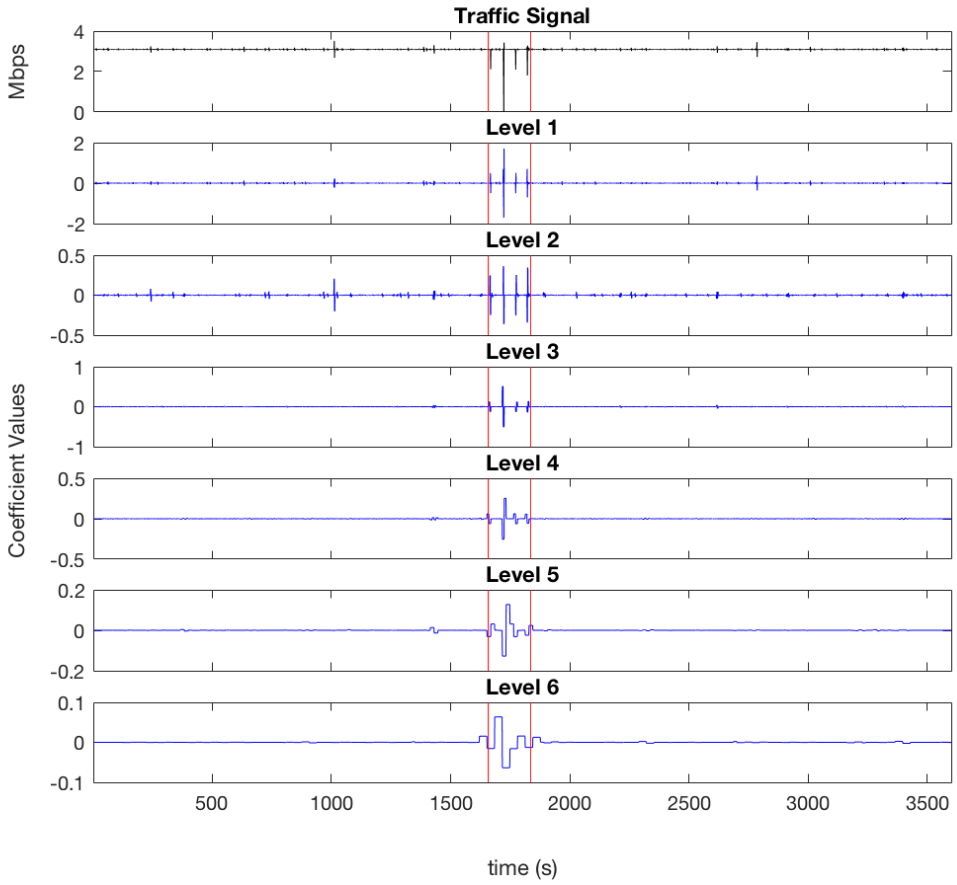

Figure 32. DDoS Attack Detection in Sent Byte Signal at Host 10.10.12.1

The anomalous traffic conditions based on the delay signal between the target host 10.10.12.1 and its intended recipient 10.10.14.1 are revealed in Figure 33. The detection method reported a single anomaly during the time interval from 1,412 to 1,640 seconds. This anomaly is consistent with the anomalous conditions reported in the other traffic signals. There were no false positives or negatives reported.

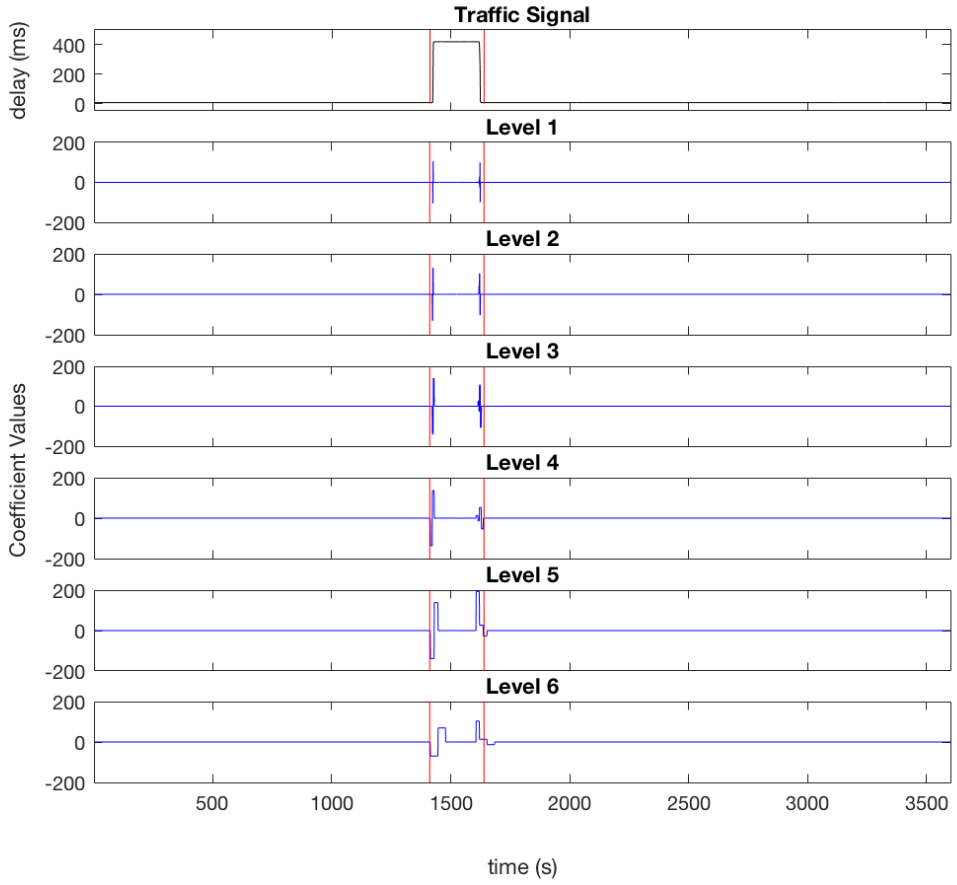

Figure 33. DDoS Attack Detection in Delay Signal at Host 10.10.12.1

c. Congestion

The anomalous traffic conditions observable in the received byte signal and delay signal collected at host 10.10.12.1 during the congestion cyber event are revealed in Figure 34 and Figure 35, respectively. The detection method accurately reported the anomalies introduced by congestion in each captured traffic signal. The effects of the congestion caused significant degradation in the received data rate and a drastic increase in the delay. The sent byte and packet signals collected at host 10.10.12.1 were also passed to the data processing and anomaly detection phases of our proposed scheme. Validating our initial observations based on the captured signals displayed in Figure 22, no anomalous conditions were reported. Since there were not any anomalous conditions

reported for the signals, their outputs from the data processing and anomaly detection phases are not displayed.

The detection method failed to detect one anomaly in the received byte signal for the congestion cyber event. The false negative is highlighted by the green box in Figure 34. The same false negative was reported in the received packet signal and is indicated in Table 5. The false negative is not considered significant because it does not cause the overall cyber event to be undetected.

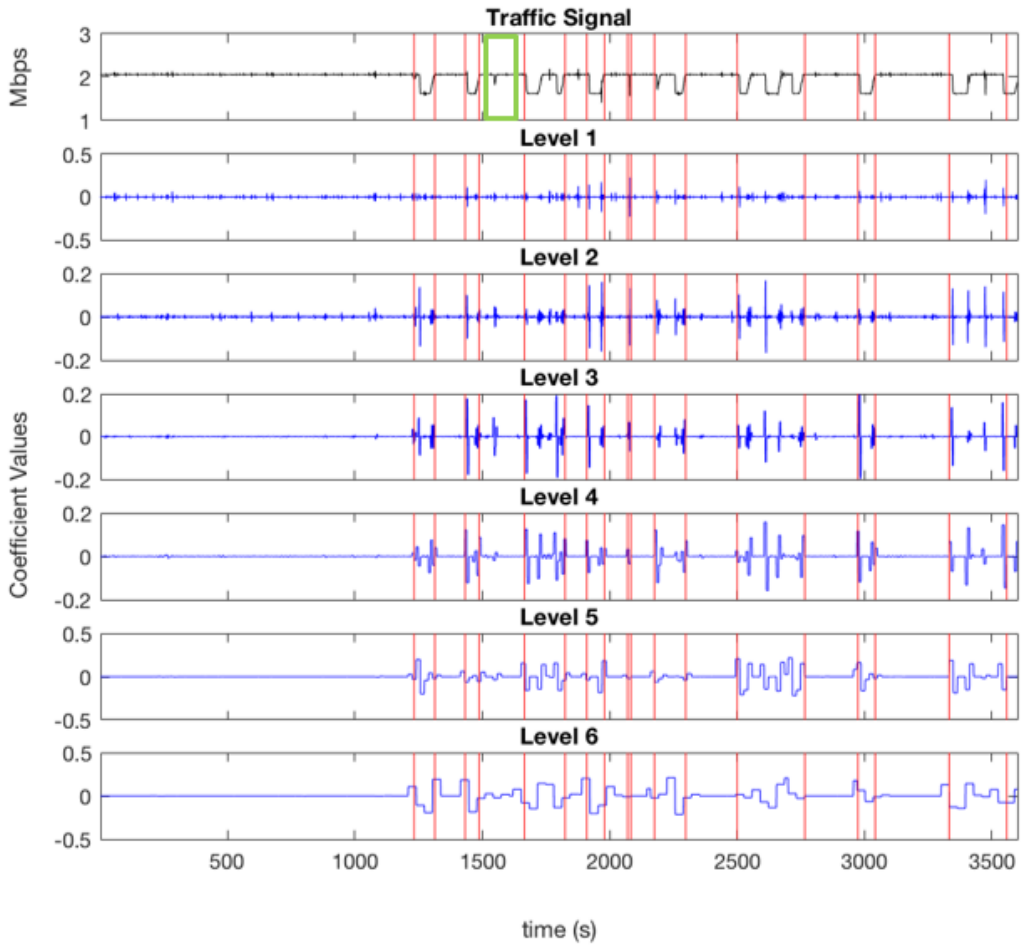

Figure 34. Congestion Detection in Received Byte Signal at Host 10.10.12.1

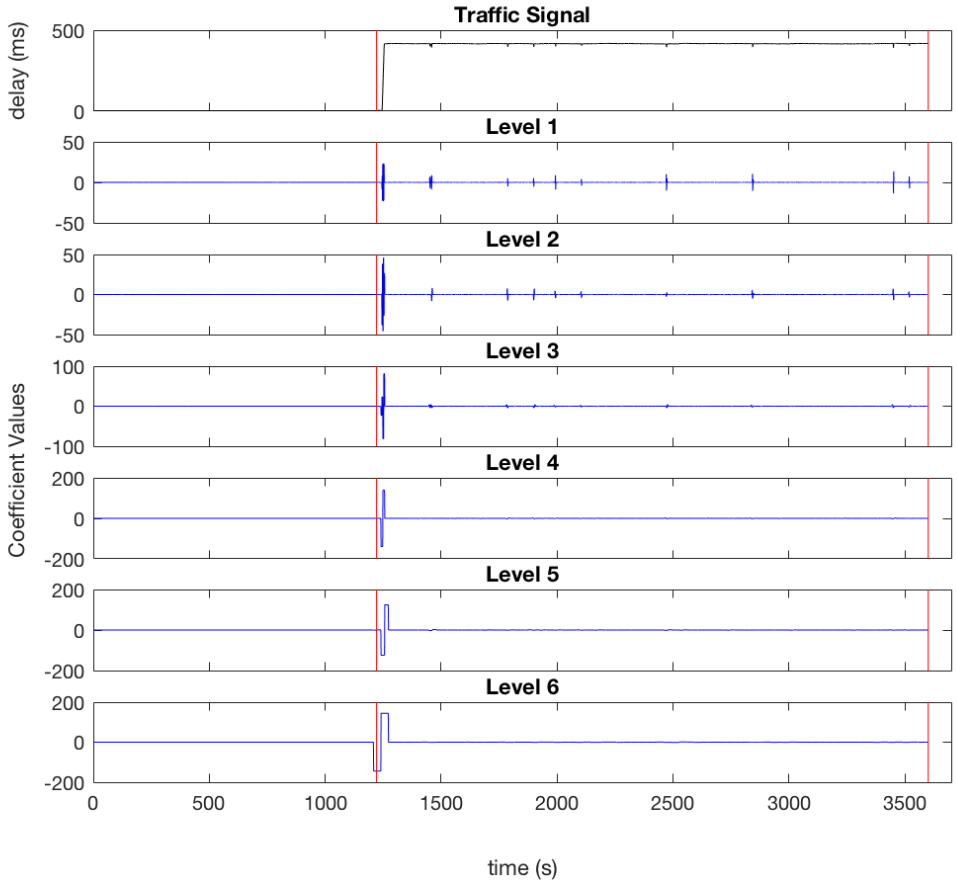

Figure 35. Congestion Detection in Delay Signal at Host 10.10.12.1

4. Cyber Event Classification Framework

We classified the cyber events in accordance with the flowchart displayed in Figure 11. The classification of the cyber events relied on the reported anomalies for the captured traffic signals displayed in Figures 29–35. The presence of reported anomalous conditions in the byte and delay signals at the analyzed ports served as the inputs to the classification flowchart. Anomalous conditions reported in the packet signals were not considered due to their mirroring of anomalous conditions reported in the byte signals. The individual paths through the classification flowchart for the anomalous conditions reported from each cyber event are displayed in Figure 36.

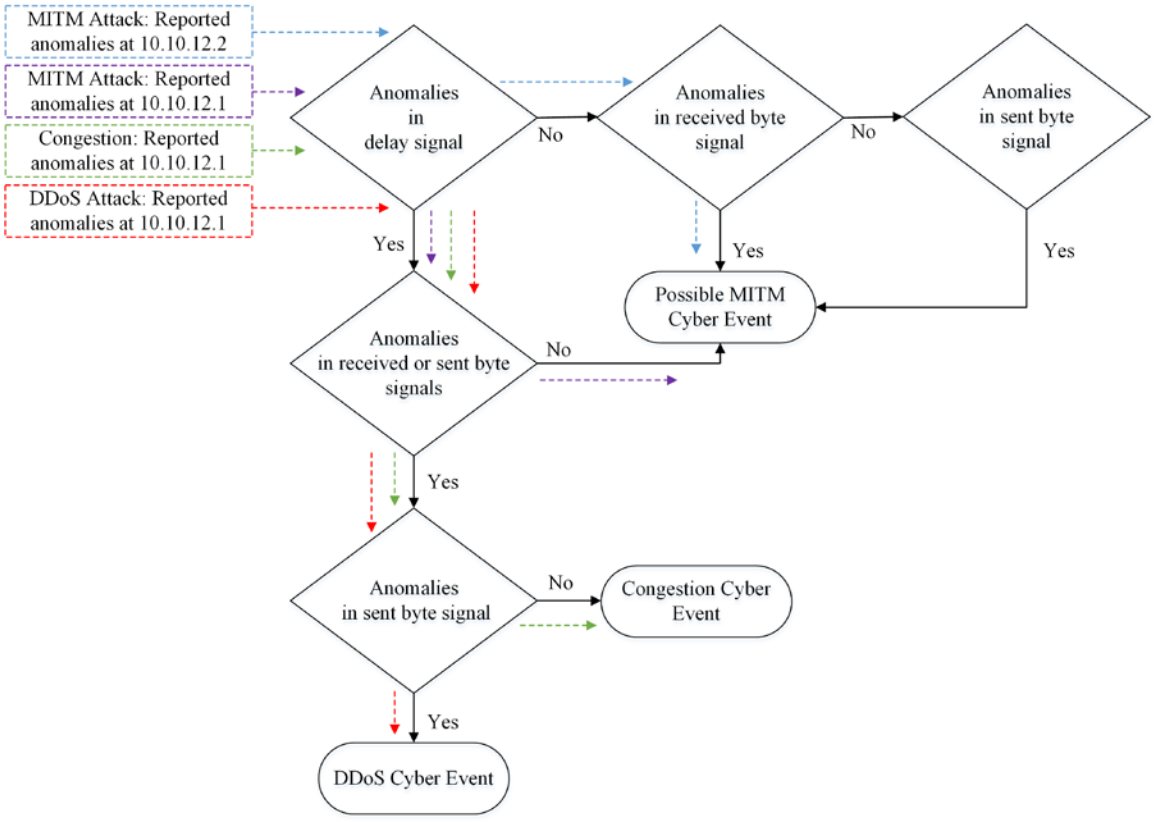

Figure 36. Cyber Event Classification

The MITM attack produced detectable anomalies in the delay signal at the switch port of the target host 10.10.12.1 and the received byte signal at the switch port of the attack host 10.10.12.2. Classification of the MITM attack was reliant on the detection of anomalous conditions at both switch ports. The presence of reported anomalies in the delay signal but not the received or sent byte signals at the target host port led to a conditional classification of the MITM attack. The path through the flowchart in Figure 36 based on reported anomalies at the target host port is highlighted by the purple arrows. The presence of reported anomalies in the received byte signal but not the delay signal at the attack host port confirmed the classification of the MITM attack. The path through the flowchart based on the anomalies reported at the attack host port is highlighted by the blue arrows.

Unlike the MITM attack, reported anomalous conditions at a single switch port were capable of classifying the DDoS attack and network congestion. The DDoS attack

produced detectable anomalies in the received byte signal, the sent byte signal, and the delay signal at the switch port for the target host 10.10.12.1. These conditions led to the classification of the DDoS attack by following the red arrows in Figure 36. Congestion produced detectable anomalies in the received byte signal and the delay signal at the switch port of host 10.10.12.1. Anomalies were not detectable in the sent byte signal of host 10.10.12.1, thereby enabling it to be differentiated from the DDoS attack. The path for classification of the congestion cyber event is represented by the green arrows in Figure 36.

5. Limitations

Application of the detection scheme validated by the results in this chapter has several key limitations related to scalability and information availability. While the controller is capable of monitoring all switch ports in the data plane to obtain the necessary byte and packet signals for the detection scheme, such a task may become untenable in a large network. Additionally, analyzing collected traffic signals for each switch port is computationally intensive. To mitigate the scalability limitations, the detection scheme could be employed selectively to switch ports determined to be particularly vulnerable to the cyber events considered. Further mitigations include variable port monitoring and adjustment of the sampling period.

Anomalous conditions reported in the captured delay signals were shown to be critical components of the classification logic. Without knowledge of the anomalous conditions in the delay signals, the cyber events are still detectable based on the byte and packet signal anomalies, but they are unable to be differentiated from each other within our classification framework. Collection of accurate delay data is a complex task. In this thesis, we collected delay data using the SPP application. Yet, this data had to be collected between two communicating hosts in the data plane; therefore, the controller had access to the byte and packet signals but not the delay signal. For application of our detection scheme at the controller, the anomalous conditions reported by the delay signals collected from hosts in the data plane would have to be transferred to the controller. This could be facilitated by the addition of a dedicated server at each data plane switch

responsible for receiving the delay signals, analyzing them for anomalous conditions, and transporting the reported information to the controller. The architecture would resemble that proposed by Ryu for the integration of Snort [24].

In this chapter, we presented the network test environment used to observe and collect traffic for the various cyber events considered, detailed our methods for emulating the cyber events, and validated our proposed scheme for anomaly detection and classification. Using wavelets to process the traffic signals for the various cyber events enabled us to clearly expose the anomalous conditions for presentation to our detection method. The multiple threshold values established through statistical analysis of normal traffic conditions enabled us to accurately report the anomalies and their associated time intervals. Unique anomalous conditions for the each cyber event enabled us to classify them. Overall, our anomaly detection and classification scheme was validated and shown to produce limited numbers of false positives and negatives. For development of the detection scheme into a deployable application, the limitations presented in this chapter must be addressed.

V. CONCLUSION

In this thesis, we used a wavelet-based detection scheme to successfully reveal anomalous conditions associated with an SDN specific MITM attack, a DDoS attack, and network congestion. Furthermore, we demonstrated that these cyber events generate unique anomalies enabling them to be differentiated and accurately classified.

A. SUMMARY OF RESULTS

The detection scheme employed in this thesis focused on revealing anomalous conditions associated with the emulated cyber events in the byte, packet, and delay traffic signals at the network layer. For each emulated cyber event, we were able to successfully collect traffic signals for byte and packet data at specified switch ports from the controller and delay data from hosts in the data plane. The collected traffic signals clearly exposed observable indicators of anomalous conditions introduced by the various cyber events. Such observable indictors validated our use of information at the network layer for anomaly detection.

Processing the captured traffic signals with the Daubechies-1 wavelet, we observed and detected periods of anomalous conditions at various wavelet levels. Our detection method experienced a limited number of false positives and negatives. The false positives and negatives were reduced by tuning the detection method for optimal window sizes and threshold values. The false negatives were experienced during detection of the network congestion. They did not hinder overall detection and classification of the network congestion cyber event. The importance of selecting the proper wavelet and detection window sizes for exposure of particular anomalies was emphasized in [8]. This was further evidenced in this thesis by varying the window size parameters.

The last phase of our detection scheme involved classification of the introduced cyber events. We presented a logic based flowchart to successfully classify the cyber events based on localization of the reported anomalies in the collected traffic signals. Each cyber event produced anomalous conditions at different ports and traffic signals.

The unique anomalous conditions reported by the detection scheme facilitated our development of the logic based classification flowchart.

Overall, we successfully satisfied our stated objectives. We presented and validated an alternate method for detection of the SDN MITM proposed by [4]. We showed that the introduction of the MITM attack diverges from normal traffic conditions and is, therefore, detectable with wavelet analysis. Furthermore, the reported anomalies associated with the attack were differentiable from other well-known cyber events detected with the scheme. By collecting and analyzing signals that were representative of traffic on the wire, we added an additional layer of defense against attackers seeking to hide. Further development of our proposed detection scheme into a deployable application is the next step to facilitate the mitigation of risks associated with SDN adoption. Such a reduction in risks is a critical requirement for the deployment of SDN in the military communications environment.

The major contributions of this thesis are represented by the methods for detection and classification of the MITM attack. To the best of our knowledge, a wavelet-based anomaly detection scheme has not been used in previous research to expose a MITM attack. Additionally, the classification of a MITM attack based on its generated traffic signals at the network layer has not been previously explored.

B. RECOMMENDATIONS AND FUTURE WORK

The primary method employed for initial detection of anomalous conditions in the reconstructed wavelet signals was based on a comparison of the calculated mean absolute deviation of samples in a detection window with their associated threshold value γ_{lev}. The threshold values γ_{lev} were established using statistical analysis of normal traffic conditions at the host ports of the switches. While this method was shown to be acceptable, it is recommended that alternate methods for threshold determination be explored and compared with the method used in this research. One such alternate method involves clustering techniques [21].

The traffic generated on the network represented UDP streams at a constant data rate. This is not reflective of the self-similar nature of Ethernet traffic. It is recommended

that more realistic traffic collected from a live local area network be introduced to the SDN environment for further validation of the detection scheme.

The complexities surrounding the collection of delay data introduce a primary shortfall of the detection scheme with regard to live deployment. Anomalous conditions exposed in the delay signals are required for accurate classification of the MITM attack and the other cyber events introduced. While the delay signals were collected using the SPP tool at the network hosts, we recommend that alternate methods be explored. Additionally, a method for transferring the delay signals to the controller is required for integration of the detection scheme into a single application.

In this thesis, the SDN MITM attack proposed by [4] was recreated in our physical SDN environment, detected, and classified. While the primary goal was to present an alternate detection method that would prevent the attackers from hiding, we were unable test the detection method under such a scenario. It is recommended that the attack be introduced with efforts for concealment in order to further validate the robustness of the detection scheme.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MITM ATTACK CODE

The key elements required to implement the MITM attack within the SDN environment consisted of the relay tunnels for the LLDP packets and the target host traffic. The relay tunnels were established between the attack hosts using the VTun protocol. The configuration files for the relay tunnels were built using the examples provided on the VTun website [27]. To relay the LLDP packets between the attack hosts, *Daemonlogger* was integrated into the VTun configuration files. The method for integration of *Daemonlogger* into the VTun configuration files was introduced by Strandboge [28]. Examples of the VTun configuration files for establishment of an LLDP relay tunnel between attack hosts 10.10.12.2 and 10.10.14.2 are displayed below. In these examples, attack host 10.10.12.2 is the server for the tunnel, and attack host 10.10.14.2 is the client. *Daemonlogger* is used at the attack host 10.10.12.2 to capture the LLDP packets from its physical interface eth0 and transfer them to its virtual tunnel interface tap0. At the attack host 10.10.14.2, *Daemonlogger* is used to capture the received LLDP packets at its virtual tunnel interface tap0 and transfer them to its physical interface eth0.

The other tunnels for relaying LLDP packets from attack host 10.10.14.2 to attack host 10.10.12.2 and redirection of target host traffic were setup with similar configuration files. The configuration files for the target traffic relay tunnel did not include *Daemonlogger*. Linux bridges were used instead of *Daemonlogger* to transfer target host traffic to the target traffic relay tunnel. An example of the bridge configuration at the attack host 10.10.12.2 is also displayed below. The bridge configuration example is based on code presented in [29]. Each code section is titled accordingly.

```
# vtund-server.conf

# Configuration file at attack host 10.10.12.2 to relay LLDP packets to
# attack host 10.10.14.2

# Define the port 10.10.14.2 should listen on for tunnel
# establishment

port 5000;
```

```
# Setup the log file for tunnel statistics
syslog      daemon;

# Define paths to required programs
ifconfig    /sbin/ifconfig;

}

# Define the default options
default {
   compress no;    # Do not use compression
   speed 0;        # Set the tunnel speed to its maximum

}

# Define the tunnel session
Attack_tunnel_1 {
passwd  Password1;    # Set the tunnel password
type  ether;          # Define the tunnel type
proto udp;            # Define the tunnel protocol
compress  no;         # Turn compression off
encrypt  no;          # Turn tunnel encryption off
stat  no;             # Turn statistics logging off
keepalive yes;        # Ensure the tunnel maintains its connection
device tap0;          # Define the virtual interface of the tunnel

# Setup the tunnel connection
up {

       # Establish the IP address for the virtual interface
       ifconfig "%% 192.168.2.1 netmask 255.255.255.0";

       # Instruct Daemonlogger to capture LLDP packets at the
       # physical interface eth0 and send them to the virtual tunnel
       # interface tap0
       program /usr/bin/daemonlogger "-f /etc/bpf.conf -i eth0 -o tap0";
   };

# Remove the tunnel interfaces and kill the Daemonlogger application
# when the tunnel session is closed
down {

# Delete the virtual tunnel interface tap0
ifconfig "tap0 down";
program /bin/ip "link delete tap0";

# Kill the Daemonlogger application
program /usr/bin/pkill "-f -x '/usr/bin/daemonlogger -f
        /etc/bpf.conf -i eth0 -o tap0'";

# Ensure all VTun sessions are closed
program /usr/bin/pkill "vtund";
    };
}
# To start VTun server session from terminal:
```

```
# vtund –n –f /etc/vtund-server.conf –s

# Create bpf.conf file in /etc directory to tell Daemonlogger to only
# forward LLDP packets received from the switch connected to attack
# host 10.10.12.2 through the LLDP relay tunnel:

# ether proto 0x88cc &&
# !(ether src f0:92:1c:22:54:bc)

**********************************************************************

# vtund-client.conf

# Configuration file at attack host 10.10.14.2 to receive LLDP packets
# from attack host 10.10.12.2 and send them to its physical interface
# eth0

Attack_tunnel_1 {
   passwd  Password1;  # Set the tunnel password
   type  ether;        # Define the tunnel type
   device tap0;        # Define the virtual interface of the tunnel

   # Setup the tunnel connection
   up {

   # Establish the IP address for the virtual interface
   ifconfig "%% 192.168.2.2 netmask 255.255.255.0";

   # Instruct Daemonlogger to capture received LLDP packets at the
   # virtual interface tap0 and send them to the physical interface
   # eth0

   program /usr/bin/daemonlogger "-f /etc/bpf2.conf -i tap0 –o
            eth0";
   };

   # Remove the tunnel interfaces and kill the Daemonlogger application
   # when the tunnel session is closed

   down {

   # Delete the virtual tunnel interface tap0
   ifconfig "tap0 down";
   program /bin/ip "link delete tap0";

   # Kill the Daemonlogger application
   program /usr/bin/pkill "-f -x '/usr/bin/daemonlogger –f
           /etc/bpf2.conf -i tap0 -o eth1'";
   };
}
# To start VTund client session from terminal:
# vtund -n -f /etc/vtund-client.conf Attack_tunnel_1 10.10.12.2


# Create bpf2.conf file in /etc directory to tell Daemonlogger to only
```

```
# forward LLDP packets to the eth0 interface:
# ether proto 0x88cc
 ***************************************************************************

# bridge-server.sh

# Linux bridge to transfer traffic from target host 10.10.12.1 to
# target traffic relay tunnel

#!/bin/bash

# Label the bridge interface and set its IP address
bridge_int="br0"
bridge_ip="0.0.0.0"

# Set the virtual interface for the bridge
tap_int="tap2"

# Specify the physical interface of the bridge
ethernet_int="eth0"
sleep 30

# Setup the bridge
brctl addbr $bridge_int

brctl addif $bridge_int $ethernet_int $tap_int
ifconfig $bridge_int $bridge_ip up
# Specify the bridge forwarding policies to only relay traffic from
# target host 10.10.12.1

ebtables -t broute -A BROUTING -p ipv4 -i eth0 --ip-dst 10.10.12.2 -j
DROP
ebtables -t broute -A BROUTING -p arp -i eth0 -d d4:be:d9:8e:60:a2 -j
DROP
ebtables -t broute -A BROUTING -p arp -i eth0 --arp-ip-dst 10.10.12.2 -
j DROP
```

APPENDIX B. MATLAB DETECTION SCHEME CODE

The primary MATLAB code sections for implementation of the anomaly detection scheme presented in Chapter III are displayed below. The separate code sections are labeled numerically. The first code section was used to establish the port specific threshold values γ_{lev} for anomaly detection. The second code section is an example implementation of the data processing and anomaly detection phases of the presented scheme. The example represents data processing and anomaly detection for the MITM attack traffic signals. The MATLAB code sections for data processing and anomaly detection of the DDoS attack and network congestion traffic signals only differ by data inputs and variable names; therefore, the specific code sections are not displayed. The functions for wavelet analysis used in the presented code sections are based on the examples presented in [17]. The code segments for the general implementation of anomaly detection and reporting are based on pseudocode from [8].

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% theshold_assignment.m
%
% Theshold assignment for anomaly detection based on normal traffic
% signals
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Import and cleanup normal traffic data
port_data = csvread('PortStats_1_Port2_Norm_05_24.csv', 1);
delay_data = csvread('N_Delay_05_24.csv');
sim_time = 7400;

[~, ~, ~, ~, P2_Bytes_Received, P2_Packets_Received, P2_Bytes_Sent,...
P2_Packets_Sent, P4_Bytes_Received, P4_Packets_Received,
P4_Bytes_Sent,...
P4_Packets_Sent,sized_delay_data] = CleanData(port_data, delay_data,...
sim_time);

% Limit data to 3,600 samples
P2_Bytes_Received = P2_Bytes_Received(end-3599:end,1);
P2_Packets_Received = P2_Packets_Received(end-3599:end,1);

P2_Bytes_Sent = P2_Bytes_Sent(end-3599:end,1);
P2_Packets_Sent = P2_Packets_Sent(end-3599:end,1);

P4_Bytes_Received = P4_Bytes_Received(end-3599:end,1);
P4_Packets_Received = P4_Packets_Received(end-3599:end,1);
```

```
P4_Bytes_Sent = P4_Bytes_Sent(end-3599:end,1);
P4_Packets_Sent = P4_Packets_Sent(end-3599:end,1);

sized_delay_data = sized_delay_data(end-3599:end,1);

analysis_time = 3600;

% Convert Byte Data to Mbps

P2_Bytes_Received = P2_Bytes_Received*8/10^6;
P2_Bytes_Sent = P2_Bytes_Sent*8/10^6;

P4_Bytes_Received = P4_Bytes_Received*8/10^6;
P4_Bytes_Sent = P4_Bytes_Sent*8/10^6;

% Convert Delay Data to ms

sized_delay_data = sized_delay_data*10^3;

%% Wavelet analysis of normal traffic data

wavelet = 'db1';
levels = 6;

% Port 2: received bytes signal

wave_data = P2_Bytes_Received;

[A_P2_Bytes_Received, D_P2_Bytes_Received, s_P2_Bytes_Received] =...
WaveletAnalysisNorm(wavelet, levels, wave_data);

% Port 2: sent bytes signal

wave_data = P2_Bytes_Sent;

[A_P2_Bytes_Sent, D_P2_Bytes_Sent, s_P2_Bytes_Sent] =...
WaveletAnalysisNorm(wavelet, levels, wave_data);

% Port 4: received bytes signal

wave_data = P4_Bytes_Received;

[A_P4_Bytes_Received, D_P4_Bytes_Received, s_P4_Bytes_Received] =...
WaveletAnalysisNorm(wavelet, levels, wave_data);

% Port 4: sent bytes signal

wave_data = P4_Bytes_Sent;

[A_P4_Bytes_Sent, D_P4_Bytes_Sent, s_P4_Bytes_Sent] =...
WaveletAnalysisNorm(wavelet, levels, wave_data);

% Port 2: received packets signal
```

```
wave_data = P2_Packets_Received;

[A_P2_Packets_Received, D_P2_Packets_Received, s_P2_Packets_Received] =
WaveletAnalysisNorm(wavelet, levels, wave_data);

% Port 2: sent packets signal

wave_data = P2_Packets_Sent;

[A_P2_Packets_Sent, D_P2_Packets_Sent, s_P2_Packets_Sent] =
WaveletAnalysisNorm(wavelet, levels, wave_data);

% Port 4: received packets signal

wave_data = P4_Packets_Received;

[A_P4_Packets_Received, D_P4_Packets_Received, s_P4_Packets_Received] =
WaveletAnalysisNorm(wavelet, levels, wave_data);

% Port 4: sent packets signal

wave_data = P4_Packets_Sent;

[A_P4_Packets_Sent, D_P4_Packets_Sent, s_P4_Packets_Sent] =
WaveletAnalysisNorm(wavelet,...
levels, wave_data);

% Port 4: delay signal

wave_data = sized_delay_data;
[A_P4_Delay, D_P4_Delay, s_P4_Delay]...
= WaveletAnalysisNorm(wavelet, levels, wave_data);

%% Assign Threshold Values (γlev)

% Multiple of standard deviation

threshold_level = 3;

% Threshold values: Port 2 - host 10.10.12.2

D_P2_Bytes_Received_reconstructed = D_P2_Bytes_Received(1,:)+
D_P2_Bytes_Received(2,:) + D_P2_Bytes_Received(3,:) +...
D_P2_Bytes_Received(4,:) + D_P2_Bytes_Received(5,:) +
D_P2_Bytes_Received(6,:);

Threshold_P2_Bytes_Received =
threshold_level*std(D_P2_Bytes_Received_reconstructed);

D_P2_Bytes_Sent_reconstructed = D_P2_Bytes_Sent(1,:)+
D_P2_Bytes_Sent(2,:) + D_P2_Bytes_Sent(3,:) +...
D_P2_Bytes_Sent(4,:) + D_P2_Bytes_Sent(5,:) + D_P2_Bytes_Sent(6,:);

Threshold_P2_Bytes_Sent =
threshold_level*std(D_P2_Bytes_Sent_reconstructed);
```

```
D_P2_Packets_Received_reconstructed = D_P2_Packets_Received(1,:)+
D_P2_Packets_Received(2,:) + D_P2_Packets_Received(3,:) +...
D_P2_Packets_Received(4,:) + D_P2_Packets_Received(5,:) +
D_P2_Packets_Received(6,:);

Threshold_P2_Packets_Received =
threshold_level*std(D_P2_Packets_Received_reconstructed);

D_P2_Packets_Sent_reconstructed = D_P2_Packets_Sent(1,:)+
D_P2_Packets_Sent(2,:) + D_P2_Packets_Sent(3,:) +...
D_P2_Packets_Sent(4,:) + D_P2_Packets_Sent(5,:) +
D_P2_Packets_Sent(6,:);

Threshold_P2_Packets_Sent =
threshold_level*std(D_P2_Packets_Sent_reconstructed);

% Threshold values: Port 4 - host 10.10.12.1

D_P4_Bytes_Received_reconstructed = D_P4_Bytes_Received(1,:)+
D_P4_Bytes_Received(2,:) + D_P4_Bytes_Received(3,:) +...
D_P4_Bytes_Received(4,:) + D_P4_Bytes_Received(5,:) +
D_P4_Bytes_Received(6,:);

Threshold_P4_Bytes_Received =
threshold_level*std(D_P4_Bytes_Received_reconstructed);

D_P4_Bytes_Sent_reconstructed = D_P4_Bytes_Sent(1,:)+
D_P4_Bytes_Sent(2,:) + D_P4_Bytes_Sent(3,:) +...
D_P4_Bytes_Sent(4,:) + D_P4_Bytes_Sent(5,:) + D_P4_Bytes_Sent(6,:);

Threshold_P4_Bytes_Sent =
threshold_level*std(D_P4_Bytes_Sent_reconstructed);

D_P4_Packets_Received_reconstructed = D_P4_Packets_Received(1,:)+
D_P4_Packets_Received(2,:) + D_P4_Packets_Received(3,:) +...
D_P4_Packets_Received(4,:) + D_P4_Packets_Received(5,:) +
D_P4_Packets_Received(6,:);

Threshold_P4_Packets_Received =
threshold_level*std(D_P4_Packets_Received_reconstructed);

D_P4_Packets_Sent_reconstructed = D_P4_Packets_Sent(1,:)+
D_P4_Packets_Sent(2,:) + D_P4_Packets_Sent(3,:) +...
D_P4_Packets_Sent(4,:) + D_P4_Packets_Sent(5,:) +
D_P4_Packets_Sent(6,:);

Threshold_P4_Packets_Sent =
threshold_level*std(D_P4_Packets_Sent_reconstructed);

D_P4_Delay_reconstructed = D_P4_Delay(1,:)+ D_P4_Delay(2,:) +
D_P4_Delay(3,:) +...
D_P4_Delay(4,:) + D_P4_Delay(5,:) + D_P4_Delay(6,:);


Threshold_P4_Delay = threshold_level*std(D_P4_Delay_reconstructed);
```

```
**********************************************************************

%% Wavelet analysis function for threshold assignment
%  Based on examples from [17]

function [A, D, s_D] = WaveletAnalysisNorm(wavelet,...
levels, wave_data)

% Initialize outputs

s_D = zeros(1,6);

% Obtain wavelet coefficients

[Coeff, P] = wavedec(wave_data, levels, wavelet);

for k = 1:levels
    A(k,:) = wrcoef('a', Coeff, P, wavelet, k);
    D(k,:) = wrcoef('d', Coeff, P, wavelet, k);
    s_D(k) = std(D(k,:));
end

**********************************************************************

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% wavelet_analysis_dection.m
%
% Wavelet analysis and detection for the MITM attack
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Import and cleanup data

port_data = csvread('PortStats_1_Port2_Fresh_05_28.csv', 1);
delay_data = csvread('LLDP_Fresh_05_28.csv');
sim_time = 4800;

[~, ~, ~, ~, P2_Bytes_Received, P2_Packets_Received, P2_Bytes_Sent,...
~, ~, ~, ~, ~, sized_delay_data] = CleanData(port_data, delay_data,...
sim_time);

% Limit data to 3,600 samples

P2_Bytes_Received = P2_Bytes_Received(end-3600:end-1);
P2_Packets_Received = P2_Packets_Received(end-3600:end-1);
sized_delay_data = sized_delay_data(end-3600:end-1);

% Convert byte data to Mbps

P2_Bytes_Received = P2_Bytes_Received*8/10^6;

% Convert delay data to ms

sized_delay_data = sized_delay_data*10^3;
```

```
analysis_time = length(P2_Bytes_Received);

%% Man-in-the-Middle Attack Analysis

% Define wavelet and detection parameters

wavelet = 'db1';
levels = 6;
window = 300;           % N_x
det_window = 10;        % N_d
level_threshold = 4;    % γ_rep
det_type = 1;
backoff = 0.2*window;   % P_b
overlap = 0.1*window;   % v

% Conduct sliding window wavelet analysis and attack detection

for flag = 1:3;

    % Port 2: wavelet analysis and anomaly detection in received byte
    % signal

    if flag == 1;

        attack_interval_Bytes = [];
        attack_interval_Bytes_transformed = [];
        attack_interval_Bytes_final = [];

        D_P2_Bytes_Received_Trace = zeros(6, analysis_time);

        % Sliding window intervals

        for u = 0:window-overlap:length(P2_Bytes_Received) - overlap;

            interval_Bytes = [];
            new_interval_Bytes = [];
            attack_interval = 0;

            % Define window start positions

            if length(P2_Bytes_Received(u+1:end)) < window;
                wave_data = P2_Bytes_Received(end-window+1:end);
                start = numel(P2_Bytes_Received(1:end-window));
            else;
                wave_data = P2_Bytes_Received(u+1:u+window);
                start = u;
            end

            threshold = Threshold_P2_Bytes_Received; % γ_lev

            % wavelet analysis and anomaly detection within specified
            % wavelet window

            [D_P2_Bytes_Received, attack_log_1,...
            attack_log_2, attack_log_3, attack_log_4, attack_log_5,...
```

```
attack_log_6] = WaveletAnalysis_Det(wavelet, levels, ...
window, wave_data, threshold, det_window, det_type, start);

if length(P2_Bytes_Received(u+1:end)) < window;
    D_P2_Bytes_Received_Trace(:,end-window+1:end) =...
    D_P2_Bytes_Received;
else;
    D_P2_Bytes_Received_Trace(:,u+1:u+window) =...
    D_P2_Bytes_Received;
end

% Initial anomolies reported at each wavelet level based on
% threshold value (γlev)

attack_log_data = [attack_log_1 attack_log_2...
attack_log_3 attack_log_4, attack_log_5, attack_log_6];

% Find repeating anomalies at multiple levels

if nnz(attack_log_data) > 0;
    remove_zeros = find(attack_log_data == 0);
    attack_log_data(remove_zeros) = [];
    possible_attack_times = unique(attack_log_data);
    multiple_level_check = histc(attack_log_data(:),...
    possible_attack_times);

    % Report repeating anomalies that meet or exceed the
    % level_threshold value (γrep)

    k = 1;
    for j = 1:length(multiple_level_check);
        if multiple_level_check(j) >= level_threshold;
            attack_interval(k) =  possible_attack_times(j);
            k = k + 1;
        end
    end

    % Define the anomalous intervals within the original
    % signal

    if attack_interval ~= 0;

        for l = 1:length(attack_interval);
            start_interval = attack_interval(l);
            end_interval = attack_interval(l) + det_window;

            interval_Bytes(l,1) = start_interval;
            interval_Bytes(l,2) = end_interval;
        end

       % Combine overlapping anomalies and report
       % anomalies within the specified wavelet window

        interval_Bytes2 = interval_Bytes;
        if numel(interval_Bytes2) > 2;
```

```
                for i = 1:length(interval_Bytes2)-1;
                    if interval_Bytes(i+1,1) -...
                       interval_Bytes(i,2) <= 0;
                       interval_Bytes2(i,2) = 0;
                       interval_Bytes2(i+1,1) = 0;
                    end
                end

                for i = 1:length(interval_Bytes2);
                    new_interval_Bytes = [new_interval_Bytes...
                    interval_Bytes2(i,:)];
                end

                discard_values = find(new_interval_Bytes == 0);
                new_interval_Bytes(discard_values) = [];

                attack_interval_Bytes =...
                [attack_interval_Bytes new_interval_Bytes];
            else
                attack_interval_Bytes =...
                [attack_interval_Bytes interval_Bytes2];
            end

        end
    end
end

% Combine anomalies not separated by the specified back-off
% period and report final anomalous intervals in the analyzed
% signal

for i = 1:2:length(attack_interval_Bytes);
    attack_interval_Bytes_transformed =...
    vertcat(attack_interval_Bytes_transformed,...
    attack_interval_Bytes(i:i+1));
end

if numel(attack_interval_Bytes_transformed) > 2;
    attack_interval_Bytes_transformed =...
    sortrows(attack_interval_Bytes_transformed);
    for i = 1:length(attack_interval_Bytes_transformed)-1;
        if attack_interval_Bytes_transformed(i+1,2) -...
            attack_interval_Bytes_transformed(i,2) <= 0;
            attack_interval_Bytes_transformed(i+1,2) =...
            attack_interval_Bytes_transformed(i,2);
        end
    end

    for i = 1:length(attack_interval_Bytes_transformed)-1;
        if attack_interval_Bytes_transformed(i+1,1) -...
            attack_interval_Bytes_transformed(i,2) <= backoff;
            attack_interval_Bytes_transformed(i,2) = 0;
            attack_interval_Bytes_transformed(i+1,1) =0;
        end
    end
```

```
        for i = 1:length(attack_interval_Bytes_transformed);
            attack_interval_Bytes_final =...
            [attack_interval_Bytes_final...
            attack_interval_Bytes_transformed(i,:)];
        end

        discard_values = find(attack_interval_Bytes_final == 0);
        attack_interval_Bytes_final(discard_values) = [];

    else;
        attack_interval_Bytes_final =...
        attack_interval_Bytes_transformed;
    end
end

% Port 2: wavelet analysis and anomaly detection in received packet
% signal

if flag == 2;

    attack_interval_Packets = [];
    attack_interval_Packets_transformed = [];
    attack_interval_Packets_final = [];

    D_P2_Packets_Received_Trace = zeros(6, analysis_time);

    % Sliding window intervals

    for u = 0:window-overlap:length(P2_Packets_Received) - overlap;
        interval_Packets = [];
        new_interval_Packets = [];
        attack_interval = 0;

        % Define window start positions

        if length(P2_Packets_Received(u+1:end)) < window;
            wave_data = P2_Packets_Received(end-window+1:end);
            start = numel(P2_Packets_Received(1:end-window));
        else;
            wave_data = P2_Packets_Received(u+1:u+window);
            start = u;
        end

        threshold = Threshold_P2_Packets_Received; % γlev

        % wavelet analysis and anomaly detection within specified
        % wavelet window

        [D_P2_Packets_Received, attack_log_1,...
        attack_log_2, attack_log_3, attack_log_4,...
        attack_log_5, attack_log_6] =...
        WaveletAnalysis_Det(wavelet, levels, window,...
        wave_data, threshold, det_window, det_type, start);
```

```
if length(P2_Packets_Received(u+1:end)) < window;
    D_P2_Packets_Received_Trace(:,end-window+1:end) =...
    D_P2_Packets_Received;
else;
    D_P2_Packets_Received_Trace(:,u+1:u+window) =...
    D_P2_Packets_Received;
end

% Initial anomalies reported at each wavelet level based on
% threshold value (γlev)

attack_log_data = [attack_log_1...
attack_log_2 attack_log_3...
attack_log_4, attack_log_5, attack_log_6];

% Find repeating anomalies at multiple levels

if nnz(attack_log_data) > 0;

    remove_zeros = find(attack_log_data == 0);
    attack_log_data(remove_zeros) = [];

    possible_attack_times = unique(attack_log_data);
    multiple_level_check = histc(attack_log_data(:),...
    possible_attack_times);

    % Report repeating anomalies that meet or exceed the
    % level_threshold value (γrep)

    k = 1;
    for j = 1:length(multiple_level_check);
        if multiple_level_check(j) >= level_threshold;
            attack_interval(k) =  possible_attack_times(j);
            k = k + 1;
        end
    end

    % Define the anomalous intervals within the original
    % signal

    if attack_interval ~= 0;
        for l = 1:length(attack_interval);
            start_interval = attack_interval(l);
            end_interval = attack_interval(l) + det_window;

            interval_Packets(l,1) = start_interval;
            interval_Packets(l,2) = end_interval;
        end
    end

    % Combine overlapping anomalies and report anomalies
    % within the specified wavelet window

    interval_Packets2 = interval_Packets;
```

```
        if numel(interval_Packets2) > 2;
            for i = 1:length(interval_Packets2)-1;
                if interval_Packets(i+1,1) -...
                   interval_Packets(i,2) <= 0;
                   interval_Packets2(i,2) = 0;
                   interval_Packets2(i+1,1) = 0;
                end
            end

            for i = 1:length(interval_Packets2);
                new_interval_Packets =...
                [new_interval_Packets interval_Packets2(i,:)];
            end

            discard_values = find(new_interval_Packets == 0);
            new_interval_Packets(discard_values) = [];

            attack_interval_Packets =...
            [attack_interval_Packets new_interval_Packets];
        else
            attack_interval_Packets =...
            [attack_interval_Packets interval_Packets2];
        end
    end
end

% Combine anomalies not separated by the specified back-off
% period and report final anomalous intervals in the analyzed
% signal

for i = 1:2:length(attack_interval_Packets);
    attack_interval_Packets_transformed =...
    vertcat(attack_interval_Packets_transformed,...
    attack_interval_Packets(i:i+1));
end

if numel(attack_interval_Packets_transformed) > 2;
    attack_interval_Packets_transformed =...
    sortrows(attack_interval_Packets_transformed);
    for i = 1:length(attack_interval_Packets_transformed)-1;
        if attack_interval_Packets_transformed(i+1,2) -...
           attack_interval_Packets_transformed(i,2) <= 0;
           attack_interval_Packets_transformed(i+1,2) =...
           attack_interval_Packets_transformed(i,2);
        end
    end

    for i = 1:length(attack_interval_Packets_transformed)-1;
        if attack_interval_Packets_transformed(i+1,1) -...
            attack_interval_Packets_transformed(i,2)<=backoff;
            attack_interval_Packets_transformed(i,2) = 0;
            attack_interval_Packets_transformed(i+1,1) =0;
        end
    end
```

```
        for i = 1:length(attack_interval_Packets_transformed);
            attack_interval_Packets_final =...
            [attack_interval_Packets_final...
             attack_interval_Packets_transformed(i,:)];
        end

        discard_values = find(attack_interval_Packets_final == 0);
        attack_interval_Packets_final(discard_values) = [];

    else;
        attack_interval_Packets_final =...
        attack_interval_Packets_transformed;
    end
end

% Port 4: wavelet analysis and anomaly detection in delay signal

if flag == 3;

    attack_interval_Delay = [];
    attack_interval_Delay_transformed = [];
    attack_interval_Delay_final = [];

    D_P4_Delay_Trace = zeros(6, analysis_time);

    % Sliding window intervals

    for u = 0:window-overlap:length(sized_delay_data) - overlap;

        interval_Delay = [];
        new_interval_Delay = [];
        attack_interval = 0;

        % Define window start positions

        if length(sized_delay_data(u+1:end)) < window;
            wave_data = sized_delay_data(end-window+1:end);
            start = numel(sized_delay_data(1:end-window));
        else;
            wave_data = sized_delay_data(u+1:u+window);
            start = u;
        end

        threshold = Threshold_P4_Delay; % γlev

        % wavelet analysis and anomaly detection

        [D_P4_Delay, attack_log_1,...
        attack_log_2, attack_log_3, attack_log_4, attack_log_5,...
        attack_log_6] =...
        WaveletAnalysis_Det(wavelet, levels, window,...
        wave_data, threshold, det_window, det_type, start);

        if length(sized_delay_data(u+1:end)) < window;
            D_P4_Delay_Trace(:,end-window+1:end) = D_P4_Delay;
```

```
else;
    D_P4_Delay_Trace(:,u+1:u+window) = D_P4_Delay;
end

attack_log_data = [attack_log_1 attack_log_2...
attack_log_3 attack_log_4, attack_log_5, attack_log_6];

% Find repeating anomalies at multiple levels

if nnz(attack_log_data) > 0;

    remove_zeros = find(attack_log_data == 0);
    attack_log_data(remove_zeros) = [];

    possible_attack_times = unique(attack_log_data);
    multiple_level_check = histc(attack_log_data(:),...
    possible_attack_times);

    % Report repeating anomalies that meet or exceed the
    % level_threshold value (γrep)

    k = 1;
    for j = 1:length(multiple_level_check);
        if multiple_level_check(j) >= level_threshold;
            attack_interval(k) =  possible_attack_times(j);
            k = k + 1;
        end
    end

    % Define the anomalous intervals within the original
    % signal

    if attack_interval ~= 0;
        for l = 1:length(attack_interval);
            start_interval = attack_interval(l);
            end_interval = attack_interval(l) + det_window;

            interval_Delay(l,1) = start_interval;
            interval_Delay(l,2) = end_interval;
        end
    end

    % Combine overlapping anomalies and report anomalies
    % within the specified wavelet window

    interval_Delay2 = interval_Delay;
    if numel(interval_Delay2) > 2;
        for i = 1:length(interval_Delay2)-1;
            if interval_Delay(i+1,1) -...
            interval_Delay(i,2) <= 0;
               interval_Delay2(i,2) = 0;
               interval_Delay2(i+1,1) = 0;
            end
        end
```

```
            for i = 1:length(interval_Delay2);
                new_interval_Delay = [new_interval_Delay...
                interval_Delay2(i,:)];
            end

            discard_values = find(new_interval_Delay == 0);
            new_interval_Delay(discard_values) = [];

            attack_interval_Delay = [attack_interval_Delay...
            new_interval_Delay];
        else
            attack_interval_Delay = [attack_interval_Delay...
            interval_Delay2];
        end
    end
end

% Combine anomalies not separated by the specified back-off
% period and report final anomalous intervals in the analyzed
% signal

for i = 1:2:length(attack_interval_Delay);
    attack_interval_Delay_transformed =...
    vertcat(attack_interval_Delay_transformed,...
    attack_interval_Delay(i:i+1));
end

if numel(attack_interval_Delay_transformed) > 2;
    attack_interval_Delay_transformed =...
    sortrows(attack_interval_Delay_transformed);
    for i = 1:length(attack_interval_Delay_transformed)-1;
        if attack_interval_Delay_transformed(i+1,2) -...
           attack_interval_Delay_transformed(i,2) <= 0;
           attack_interval_Delay_transformed(i+1,2) =...
           attack_interval_Delay_transformed(i,2);
        end
    end

    for i = 1:length(attack_interval_Delay_transformed)-1;
        if attack_interval_Delay_transformed(i+1,1) -...
            attack_interval_Delay_transformed(i,2) <= backoff;
            attack_interval_Delay_transformed(i,2) = 0;
            attack_interval_Delay_transformed(i+1,1) =0;
        end
    end

    for i = 1:length(attack_interval_Delay_transformed);
        attack_interval_Delay_final =...
        [attack_interval_Delay_final...
        attack_interval_Delay_transformed(i,:)];
    end

    discard_values = find(attack_interval_Delay_final == 0);
    attack_interval_Delay_final(discard_values) = [];
else
```

```
            attack_interval_Delay_final =...
            attack_interval_Delay_transformed;
        end
    end
end

**************************************************************************

% Wavelet analysis and anomaly detection function

function [D_Trace, attack_log_1, attack_log_2, attack_log_3,...
attack_log_4,           attack_log_5,           attack_log_6]          =
WaveletAnalysis_Det(wavelet, levels, window,...
wave_data, threshold, det_window, det_type, start)

% Initialize outputs

attack_log_1 = 0;
attack_log_2 = 0;
attack_log_3 = 0;
attack_log_4 = 0;
attack_log_5 = 0;
attack_log_6 = 0;

m = 1;
n = 1;
o = 1;
p = 1;
s = 1;
v = 1;

% Conduct wavelet decomposition

[Coeff, P] = wavedec(wave_data, levels, wavelet);

% Conduct wavelet reconstruction

for k = 1:levels
    D(k,:) = wrcoef('d', Coeff, P, wavelet, k);
end

D_Trace = D;

% Anomaly detection with detection window

for l = 1:levels;

    % Detection with mean absolute deviation

    if det_type == 1;

        for q = 0:window - det_window

            mean_window = mad(D(l,q+1:q+det_window));
```

```
            if mean_window > threshold;
                if l == 1;
                    attack_log_1(m) = start+q+1;
                    m = m+1;
                elseif l == 2;
                    attack_log_2(n) = start+q+1;
                    n = n+1;
                elseif l == 3;
                    attack_log_3(o) = start+q+1;
                    o = o+1;
                elseif l ==4;
                    attack_log_4(p) = start+q+1;
                    p = p+1;
                elseif l == 5;
                    attack_log_5(s) = start+q+1;
                    s = s+1;
                else;
                    attack_log_6(v) = start+q+1;
                    v = v+1;
                end
            end
        end
    end
end
```

LIST OF REFERENCES

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," in *Proc. of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[2] Y. Jarraya, T. Madi and M. Debbabi, "A Survey and a Layered Taxonomy of Software-Defined Networking," *IEEE Commun. Surveys Tut.*, vol. 16, no. 4, pp. 1955–1980, Apr. 2014.

[3] M. Lawlor. (2008, Jul. 15). NMCI in a box. [Online]. Available: http:// www.afcea.org/content/?q=nmci-box

[4] S. Hong, L. Xu, H. Wang and G. Gu, "Poisoning Network Visibility in Software Defined Networks: New Attacks and Countermeasures," in *Proc. Network and Distributed System Security Symp.*, San Diego, CA, 2015.

[5] L. B. Kish, "Protection Against the Man-in-the-Middle-Attack for the Kirchhoff-Loop Johnson(-like)-Noise Cipher and Expansion by Voltage-Based Security," *Fluctuation and Noise Letters*, vol. 6, no. 1, pp. L57-L63, 2006.

[6] P. Barford, J. Kline, D. Plonka and A. Ron, "A Signal Analysis of Network Traffic Anomalies," in *Proc. of the 2nd ACM SIGCOMM Workshop on Internet Measurement*, Marseille, France, 2002, pp. 71–82.

[7] C. T. Huang, S. Thareja and Y. J. Shin, "Wavelet-based Real Time Detection of Network Traffic Anomalies," *Int. J. of Network Security*, vol. 6, no. 3, pp. 309–320, May 2008.

[8] S. S. Kim and A. L. N. Reddy, "Statistical Techniques for Detecting Traffic Anomalies Through Packet Header Data," *IEEE/ACM Trans. Netw.*, vol. 16, no. 3, pp. 562–575, Jun. 2008.

[9] X. Wang, N. Gao, L. Zhang, Z. Liu and L. Wang, "Novel MITM Attacks on Security Protocols in SDN: A Feasibility Study," in *Proc. 18th Int. Conf. Inform. Commun. Security*, Singapore, 2016, pp. 455–465.

[10] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting Security Attacks in Software-Defined Networks," in *Proc. Network and Distributed System Security Symp.*, San Diego, CA, 2015.

[11] A. Lakhina, M. Crovella, and C. Diot, "Characterization of Network-Wide Anomalies in Traffic Flows," in *Proc. 4th ACM SIGCOMM IMC*, Taormina, Italy, 2004, pp. 201–206.

[12] Open Networking Foundation (ONF), “OpenFlow Switch Specification,” Tech. Rep., Feb. 2011. [Online]. Available: http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf

[13] F. Pakzad, M. Portmann, W. L. Tan and J. Indulska, “Efficient Topology Discovery in Software Defined Networks,” in *Proc. 8th Int. Conference on Signal Processing and Communication Systems*, Gold Coast, QLD, 2014, pp. 1–8.

[14] S. T. Zargar, J. Joshi and D. Tipper, “A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks,” *IEEE Commun. Surveys Tut.*, vol. 15, no. 4, pp. 2046–2069, Mar. 2013.

[15] RioRay. (2015). Taxonomy of DDoS attacks. [Online]. Available: http://www.riorey.com/types-of-ddos-attacks/

[16] The Linux Information Project. (2005, Nov 28). Network congestion definition. [Online]. Available: http://www.linfo.org/congestion.html

[17] M. Misiti, Y. Misiti, G. Oppenheim and J. Poggi, “Wavelet Toolbox – User’s Guide,” The MathWorks Inc., Version 1, 1996.

[18] S. Mallat, “Wavelet Bases,” in *A Wavelet Tour of Signal Processing*, 2nd ed. San Diego, CA, USA: Academic Press, 1999, ch. 2, sec. 3, pp. 255–265.

[19] S. Zander and G. Armitage, “Minimally-Intrusive Frequent Round Trip Time Measurements Using Synthetic Packet-Pairs - Extended Report,” Centre for Advanced Internet Architectures, Melbourne, Australia, Tech. Rep. 130730A, Jul. 2013.

[20] D. Roberts and F. Roberts. (2017). Mean absolute deviation. [Online]. Available: https://mathbits.com/MathBits/TISection/Statistics1/MAD.html

[21] V. Chandola, A. Banerjee and V. Kumar, “Anomaly Detection: A Survey,” *ACM Computing Surveys*. Vol. 41, no. 3, Article 15, Jul. 2009.

[22] Nippon Telegraph and Telephone Corporation, Tokyo, Japan. (2017). *RYU*. [Online]. Available: https://github.com/osrg/ryu. Accessed: Jul. 25, 2017.

[23] RYU Project Team. (2014). *RYU SDN Framework – English Edition Release 1.0*. [Online]. Available: https://osrg.github.io/ryu-book/en/Ryubook.pdf

[24] Nippon Telegraph and Telephone Corporation. (2014). *RYU the Network Operating System (NOS)*. [Online]. Available: http://ryu.readthedocs.io/en/latest/. Accessed: Jul. 25, 2017.

[25] University of Illinois. (2010). *iperf2*. [Online]. Available: https://sourceforge.net/projects/iperf2/. Accessed: Jul. 25, 2017.

[26] M. Roesch. (2016). *Daemonlogger*. [Online]. Available: https://sourceforge.net/projects/daemonlogger/. Accessed: Jul. 25, 2017.

[27] M. Krasnyansky. (2016). *VTun 3.0.4.* [Online]. Available: http://vtun.sourceforge.net/. Accessed: Jul. 25, 2017.

[28] J. Strandboge. (2013, Jan. 15). Mirror, mirror. [Online]. Available: https://penguindroppings.wordpress.com/2013/01/15/mirror-mirror/

[29] OpenVPN Technologies, Inc. (2008). Bridging overview. [Online]. Available: https://openvpn.net/index.php/open-source/documentation/miscellaneous/76-ethernet-bridging.html

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California